Planarity Testing and Embedding Algorithms

by

D.I. Carson

Submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in the
Department of Computer Science
University of Natal, Durban
1990

Preface

This research was carried out in the Department of Computer Science at the University of Natal, Durban from April 1989 to December 1990 under the supervision of Dr O.R. Oellermann.

This thesis represents original work by the author, and has not been submitted in part, or in whole, to any other university. Where use is made of the work of others, it has been duly acknowledged in the text.

Acknowledgements

My sincere thanks go to my supervisor, Dr O.R. Oellermann, for the continuing support and assistance during my degree.

Thanks go to my able team of proof-readers Nancy Odendaal and Ethel Carte.

Finally, I wish to thank the Foundation for Research Development and the University of Natal for the financial assistance to enable me to pursue this degree.

Abstract

This thesis deals with several aspects of planar graphs, and some of the problems associated with non-planar graphs.

Chapter 1 is devoted to introducing some of the fundamental notation and tools used in the remainder of the thesis.

Graphs serve as useful models of electronic circuits. It is often of interest to know if a given electronic circuit has a layout on the plane so that no two wires cross. In Chapter 2, three efficient algorithms are described for determining whether a given 2-connected graph (which may model such a circuit) is planar. The first planarity testing algorithm uses a path addition approach. Although this algorithm is efficient, it does not have linear complexity. However, the second planarity testing algorithm has linear complexity, and uses a recursive fragment addition technique. The last planarity testing algorithm also has linear complexity, and relies on a relatively new data structure called PQ-trees which have several important applications to planar graphs. This algorithm uses a vertex addition technique.

Chapter 3 further develops the idea of modelling an electronic circuit using a graph. Knowing that a given electronic circuit may be placed in the plane with no wires crossing is often insufficient. For example, some electronic circuits often have in excess of 100 000 nodes. Thus, obtaining a description of such a layout is important. In Chapter 3 we study two algorithms for obtaining such a description, both of which rely on the PQ-tree data structure. The first algorithm determines a rotational embedding of a 2-connected graph. Given a rotational embedding of a 2-connected graph, the second algorithm determines if a convex drawing of a graph is possible. If a convex drawing is possible, then we output the convex drawing.

In Chapter 4, we concern ourselves with graphs that have failed a planarity test of Chapter 2. This is of particular importance, since

complex electronic circuits often do not allow a layout on the plane. We study three different ways of approaching the problem of an electronic circuit modelled on a non-planar graph, all of which use the PQ-tree data structure. We study an algorithm for finding an upper bound on the thickness of a graph, an algorithm for determining the subgraphs of a non-planar graph which are subdivisions of the Kuratowski graphs K_5 and $K_{3,3}$, and lastly we present a new algorithm for finding an upper bound on the genus of a non-planar graph.

Table of Contents

Chapter 1
Introductory Topics 1
Section 1.1
Graph Planarity 2
Section 1.2
Complexity Theory 6
Section 1.3
The Depth-First Search 9
Section 1.4
The Rotational Embedding Scheme 14
Section 1.5
Graph Data Structures 20
Chapter 2
Planarity Testing Algorithms 27
Section 2.1
Path Addition Algorithm 29
Section 2.2
Fragment Addition Algorithm 42
Section 2.3
Vertex Addition Algorithm 65
Section 2.4
PQ-trees and a Linear Vertex Addition Algorithm 77
Section 2.5
Implementation of PQ-trees in linear time 101
Chapter 3
Drawing a Graph 130
Section 3.1
Finding an Embedding of a planar graph 135
Section 3.2
Triconnectivity Testing by Path Addition 152
Section 3.3
Triconnectivity Testing by PQ-Trees 174
Section 3.4
Drawing a Graph in the plane 195
Chapter 4
Non-Planar Graphs 225
Section 4.1
The Maximum Planar Subgraph Problem 227
Section 4.2
Finding the Obstructions to Planarity 256
Section 4.3
A New Algorithm for finding an Upper Bound of the Genus of a Graph 278
References 293
Appendix A
Source Code Listings 298

Chapter 1

Introductory Topics

In this chapter we introduce some of the tools necessary for the ensuing chapters. In Section 1.1 we cover the graph theory and the fundamentals of planar graphs which we use throughout this thesis. Section 1.2 describes some aspects of complexity theory used throughout this thesis. In Section 1.3, we describe the Depth-First Search (DFS) on a graph, as well as some important properties of this search. The DFS of a graph is an integral part of many of the algorithms explored later, and has several important properties. Section 1.4 is devoted to the description of graph embeddings. In Section 1.5, we discuss the computer implementation of graph algorithms. In particular, we shall examine data structures used in many of the algorithms covered in this thesis.

Section 1.1

Graph Planarity

In this section we look at various well-known results in graph theory, in particular as they pertain to planar graphs. We follow Behzad, Chartrand and Lesniak-Foster [BCL79], Chiba and Nishizeki [CN88] and Chartrand and Oellermann [CO90] for some details. If a graph G has $|E(G)| = q$, and $|V(G)| = p$, then we say G is a (p, q) graph. The only non-standard notation used in this thesis, is that arcs are referred to as directed edges, or, where there is no ambiguity, simply as edges.

We have the following definition from [BCL79]. A graph G is said to be *realisable* or *embeddable* on a surface S, if it is possible to distinguish a collection of p distinct points of S which correspond to the vertices of G, and a collection of q curves which correspond to the edges of G, which are pairwise disjoint except possibly for the endpoints on S, such that if a curve A corresponds to the edge e = uv, then only the endpoints of A correspond to vertices of G, namely u and v.

Following from this definition, we say that a graph G is *planar* if it is embeddable in the plane. If a planar graph G is embedded in the plane, then we say that the embedding is a *plane* graph. We say the plane graph is a *planar realisation* of the planar graph G. Given a plane graph G, a *region* of G is a maximal portion of the plane for which any two points may be joined by a curve A which does not intersect any curve corresponding to an edge of G, or any point corresponding to a vertex of G. The *outer region* is the unbounded region on the plane.

There is a well-known, and simple, formula by Euler (1750), that relates the number of vertices, edges and regions of a graph G.

Theorem 1.1: Let G be a connected plane (p, q) graph, with and r regions. Then

$$p - q + r = 2.$$

A graph G is *maximal planar* if G is planar, and G + uv is non-planar for every pair u, v of vertices of G. From [BCL79], we have the following corollary of Theorem 1.1.

Corollary 1.1: If G is a maximal planar (p, q) graph, then $q = 3p - 6$.

Since every planar graph is a subgraph of some maximal planar graph, we obtain the following result.

Corollary 1.2: If G is a planar (p, q) graph, then $q \leq 3p - 6$.

Corollary 1.2 is a simple, yet important result. For the planarity testing algorithms of Chapter 2, it suffices to consider (p, q) graphs with $q \leq 3p - 6$, for, if this bound is not satisfied, then, by Corollary 1.2, the graph is automatically non-planar.

We now consider some fundamental results on planar graphs. We define a graph G' to be a *subdivision* of a graph G, if G' is obtained from G, by repeatedly replacing edges of G by paths having the same endpoints as the edges. We have the following well-known planarity testing result by Kuratowski [Kur30].

Theorem 1.2: A graph G is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$ as a subgraph.

We call K_5 and $K_{3,3}$ the Kuratowski graphs.

Let H be a subgraph of G. We define a relation ~ on E(G) - E(H) as follows: if $e, f \in E(G) - E(H)$, then $e \sim f$ if and only if there exists a walk W in G - E(H) whose first and last edges are e and f respectively, and no internal vertex of W belongs to V(H). We note that ~ is an equivalence relation on E(G) - E(H). The subgraphs induced by the equivalence classes of ~ on

E(G) - E(H) are called *fragments* of G with respect to H. If $\mathcal{F}$ is a fragment of G with respect to H, then the vertices in $V(\mathcal{F}) \cap V(H)$ are called the *attachments* of $\mathcal{F}$ on H.

Now, suppose C is a cycle in a 2-connected graph G. We say that two fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ of G with respect to C interlace if at least one of the following conditions holds.

(a) There are distinct attachments u and v of $\mathcal{F}_1$, and w and x of $\mathcal{F}_2$, such that on C they appear in the order u, w, v, x.
(b) There are three attachments common to $\mathcal{F}_1$ and $\mathcal{F}_2$.

Intuitively, if G is embeddable in the plane, then, if two fragments interlace, they may not be drawn on the same side of C, in a plane embedding of G.

If $\mathcal{F}$ is a fragment of G with respect to C, then $C + \mathcal{F}$ is the subgraph of G induced by the edges of $E(C) \cup E(\mathcal{F})$. We have the following important theorem, from Even [Eve79].

Theorem 1.3: Let G be a 2-connected graph, and C some cycle of G. Let $\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_m$ be the fragments of G with respect to C. Then, G is planar if the following two conditions hold :

(a) For every fragment $\mathcal{F}_i$, $C + \mathcal{F}_i$ is planar $(1 \leq i \leq m)$.
(b) The set F of fragments may be partitioned into two subsets, such that no two fragments in the same subset interlace.

Proof: Let the two subsets of F referred to in (b) be A and B. Assume that $|A| \neq \emptyset$. Observe that there must be a fragment $\mathcal{F} \in A$, such that, if we proceed along C, then we encounter all the attachments of $\mathcal{F}$ without encountering an attachment w of any other fragment $\mathcal{F}_k \in A$ such that w is not an attachment of $\mathcal{F}$.

To see this, proceed along the cycle, starting at some attachment v of some fragment $\mathcal{F}_i \in A$. If some attachment w of another fragment $\mathcal{F}_j \in A$, which is not an attachment of $\mathcal{F}_i$, is encountered before we encounter all the attachments of $\mathcal{F}_i$, then we restart the process, starting with w and $\mathcal{F}_j$. Since no two of the fragments in A interlace, and since there are a

finite number of fragments, we must eventually find a fragment $\mathcal{F}_k$ so that if we proceed along C, then we encounter all the attachments of $\mathcal{F}_k$ without encountering an attachment x of any other fragment $\mathcal{F}$ of A such that x is not an attachment of $\mathcal{F}_k$.

Now, we embed G in the plane as follows. Place C in the plane. Let the first and last vertices of $\mathcal{F}_k$ encountered, as we proceed along C, be v_f and v_ℓ, respectively. We place $\mathcal{F}_k$ inside C. Note that, by condition (a), C + $\mathcal{F}_k$ is planar. Let P be the path, with start vertex v_f and end vertex v_ℓ, which we just traversed to encounter the attachments of $\mathcal{F}_k$. Now, consider the cycle $C' = C - V(P) \cup v_f v_\ell$, and the set $A - \{\mathcal{F}_k\}$. We may repeat the process of selecting a fragment $\mathcal{F}$ and modifying C to place all the fragments $\mathcal{F}_i \in A$ inside the cycle C. A similar argument holds for all fragments $\mathcal{F}_j \in B$, except, we place those fragments outside C. Thus, G is planar. ❑

Theorem 1.3 is an important result, that forms the basis for one of the planarity testing algorithms we study in Chapter 2.

Section 1.2

Complexity Theory

As a guide for this section, we use Chartrand and Oellermann [CO90] and Althoen and Bumcrot [AB88].

The *complexity* of an algorithm measures the amount of computational effort expended when the computer solves a problem using that algorithm. The measure may refer to the number of computational steps, the running time or storage space required. The complexity of an algorithm is typically a function of the size and presentation of the input data.

Consider the following example. Suppose that there are two algorithms to find the inverse of a matrix. Suppose that Algorithm A will find the inverse of an n x n matrix in at most n^4 units of time. Further, suppose that another algorithm, Algorithm B say, will find the inverse of an n x n matrix in at most $4n^3$ units of time. Now, to find the inverse of a 4 x 4 matrix, Algorithm A will take 256 units of time, and Algorithm B also takes 256 units of time. However, for $n > 4$, Algorithm B is faster than Algorithm A, but for $n < 4$, Algorithm A is faster than Algorithm B.
Note that both Algorithm A and Algorithm B require at most n^4 and $4n^3$ units of time respectively. This kind of complexity measure we call *worst case complexity*. For the rest of this thesis we shall concentrate on worst case time complexity of algorithms.

Although worst case complexity is the most common form of complexity measure, there are other measures. For example, average case complexity describes the average running time of an algorithm over all possible data inputs. The average case complexity measure is often difficult to quantify. Firstly, there may be a very large set of possible data inputs. Thus, it may not be feasible to time the running of an algorithm over all possible data inputs, and we are forced to approximate the

average case complexity. Secondly, the average case complexity should also consider the frequency with which a certain class of data inputs is used. Suppose that we wish to determine the average case complexity of some algorithm A. Further, suppose that there are 100 possible data inputs for algorithm A. Now, if there is a large probability that a data input from a subset, say of 25 of the possible data inputs, will be used as input to algorithm A, then the average case complexity of algorithm A should reflect a bias towards the complexity of A when those 25 data inputs are used. In practice it is sometimes very difficult to accurately describe the input, and hence it is very difficult to be able to correctly compute the average case complexity.

We say an algorithm is *efficient* if its complexity is bounded by a polynomial in the input size n. For example, an algorithm with complexity $\sqrt{n}$ is efficient, but another algorithm with complexity 2^n is not. A computational problem is called *tractable* if there exists an efficient algorithm for solving the problem. Similarly, a computational problem is called *intractable* if it can be established that there exists no efficient algorithm for solving the problem.

To compare two algorithms more effectively, we introduce the following notation. Suppose that we express the complexity of an algorithm as a function f(n) of the size n of the data input. A function f(n) is said to be of the *order of magnitude* g(n), if there are constants N and k such that

$$\text{for all } n \geq N,\ f(n) \leq k\, g(n)$$

We write f(n) = O(g(n)). Suppose Algorithm A has complexity f(n) and Algorithm B has complexity g(n). If f(n) = O(g(n)) but g(n) ≠ O(f(n)), then we say that Algorithm A is more efficient than Algorithm B. If f(n) = O(g(n)) and g(n) = O(f(n)), we say that Algorithms A and B have the same complexity.

Listed below, in increasing order of complexity, are some of the more common functions that g(n), and hence an algorithm's complexity, may be written as.

$$O(\log n), O(n), O(n \log n), O(n^2), ..., O(2^n)$$

In this thesis, we are particularly interested in algorithms with *linear* or O(n) complexity.

There are many problems for which it is not known whether the given problem is tractable or intractable. Among these problems is the class of NP-complete problems. These problems are equivalent, in the sense that an efficient algorithm which solves one of the problems guarantees, for every other NP-complete problem, an efficient algorithm which solves it. It is believed by many experts in the field that the class of NP-complete problems is intractable. Thus, trying to find an efficient algorithm to solve one of them should receive a lower priority than other approaches. If a problem is an NP-complete problem, then, perhaps, we should rather concentrate on heuristics to approximate the solution to within an "acceptable limit". An entire book has been devoted to the study of NP-complete problems [GJ79].

Section 1.3

The Depth-First Search

The Depth-First Search (DFS) is a very important tool in graph algorithms, and in particular in algorithms pertaining to planar graphs. We shall use the DFS in the form given by Hopcroft and Tarjan ([HT73a], [HT72]). In order to describe a DFS of a graph, we need the concept of a directed and rooted tree. A tree T is a *directed tree* if T is obtained from a tree by assigning directions to its edges. A *rooted tree* T is a directed tree which contains a vertex v, called the *root*, such that for every vertex x of T, there exists a directed v-x path in T. The vertex which precedes x on this v-x path is called the *parent* of x in T. If T is a rooted tree, and w is a vertex of T, then the *subtree rooted at* w is the maximal subtree T' of T which is rooted at w.

A DFS of a graph G constructs a directed DFS spanning forest F of G. In fact, every component T of F will be a rooted tree. To construct F we need to store three quantities about the edges and vertices. First of all, at any stage of the DFS we need to know if a particular edge has been used or not. Initially all edges are marked unused. Then, we mark each edge used once the DFS has scanned that edge. Secondly, for every vertex $v \in V(G)$ we store a value Parent(v) which gives the parent of v in the directed tree T. The root of T, denoted by Root(T), is the unique vertex u of T which has no parent. Lastly, for every vertex $v \in V(G)$, we need to store a unique number called the DFS index of v. We denote the DFS index of a vertex v by dfi(v). If dfi(v) = i, then v is the i-th vertex which is visited during a DFS.

Briefly, the algorithm works, for a component of G, as follows. Initially all edges are unused and all vertices have no DFS index. We arbitrarily choose a starting vertex, s say, and make s the vertex we are currently visiting. We assign s a DFS index of 1. Suppose our current vertex is a vertex u. We proceed as follows.

Case 1: If no unused edge e = uv incident with u exists then we have two subcases. If we are at the root of T, we stop with the DFS completed. Otherwise, we backtrack to the parent of u in T and let it be the current vertex.

Case 2: If an unused edge e = uv exists, we mark e used. If v has a DFS index, then we go back and consider another edge incident with u. Otherwise, if v has no DFS index, we add this edge, directed from u to v, to our DFS tree T. We assign Parent(v) = u, give v the next available label as DFS index, set u = v and loop back to consider the edges incident with v.

Algorithm 1.1 gives the full DFS algorithm.

Algorithm 1.1: Depth-First Search

{ Perform a DFS on an arbitrary graph }

```
For every edge e = uv do
    mark e unused
For every vertex v ∈ V(G) do
    dfi(v) = 0                  { Give them no DFS index }
Current_Label = 1       { First DFS index to assign }

repeat                          { for every component of G }
    Start_Vertex = a vertex v ∈ V(G) with dfi(v) = 0
    u = Start_Vertex            { start vertex is the root of the component }
    dfi(u) = Current_Label      { assign DFS index of Root(T) }
    Current_Label = Current_Label + 1

    Finished = false        { to indicate when we have completed }
                            { DFS for this component }
    repeat                              { perform the DFS }
        if there are no unused edges incident with u
            then                        { we backtrack if possible }
                if u = Start_Vertex
                    then Finished = true        { we are at the root }
                    else u = Parent(u)          { else backtrack }

            else    { there is an unused edge }
                Choose an unused edge e = uv and direct the edge u --e--> v
                Mark e used
                if dfi(v) = 0       { check if it is an edge of DFS tree T }
                    then            { If it is then ... }
                        Parent(v) = u   { Make it a child of u }
                        u = v           { Make it new current vertex }
                        dfi(u) = Current_Label
                                        { and give it a DFS label }
```

Current_Label = Current_Label + 1
until Finished { when we reach root again and finish all incident edges }

until for all v ∈ V(G), dfi(v) ≠ 0 { until we have done every component }

end

For a given (p, q) graph G, there are a number of effects that the DFS has on G. Firstly, every vertex v ∈ V(G) obtains a unique Depth-First index dfi(v), where dfi(v) ∈ {1, 2, ..., p}. The DFS creates a digraph D from G by assigning a direction to each edge e ∈ E(G). Note that, for a component H of G, Algorithm 1.1 does not produce a spanning DFS tree T. The tree T is an underlying digraph of the corresponding component D' of D, called the DFS tree of H. We define an edge $u \xrightarrow{e} v$ of D' to belong to E(T) if dfi(u) < dfi(v). Such an edge e we call a *tree edge* of D'. All edges of E(D') - E(T) we call *back edges* of D'.

Lemma 1.1: The complexity of Algorithm 1.1 is O(max {p, q}).
Proof: Apart from some initialisation steps, in the algorithm, we traverse each edge exactly once, and visit every vertex v at most once for each vertex adjacent to v. Thus, Algorithm 1.1 has complexity O(max (p, q)), and the lemma is proved. ❑

If G is a planar (p, q) graph, then, from Corollary 1.2, $q \leq 3p - 6$, and so in this case, Algorithm 1.1 has complexity O(p). Using Algorithm 1.1 we may easily determine the components of a graph G. For the rest of this thesis, we assume that G is connected.

If $u \xrightarrow{e} v$ is a tree edge, we say that v is a *child* of u. We define an *ancestor* v of a vertex w to be a vertex on the unique path of tree edges from the root of T to w. We say that w is a *descendant* of v.

Lemma 1.2: Let D be the digraph produced by Algorithm 1.1, and T the corresponding DFS tree. If $u \xrightarrow{e} v$ is a back edge of E(D), then v is an ancestor of u in T.
Proof: Note that, since e is a back edge, dfi(u) > dfi(v). Thus, v is explored before u in the DFS. The lemma follows from the fact that we only

backtrack from a vertex when all its incident edges have been scanned. If u is not a descendant of v, then the DFS would backtrack to an ancestor of v before u is scanned. But, we would scan edge e before backtracking from v. Thus, u is a descendant of v. ❑

From Lemma 1.2, we may observe that, during a DFS, all ancestors of a vertex v have been visited before we visit v for the first time, and that back edges must proceed from a descendant to an ancestor.

Define S(v) to be the set of vertices reachable from v by a (non-trivial) path of tree edges followed by at most one back edge. Then, for a vertex $v \in V(G)$, we define the *first lowpoint* of a vertex v of D, denoted $L_1(v)$, as

$$L_1(v) = \min \{ \{dfi(v)\} \cup \{dfi(u) \mid u \in S(v)\} \}$$

Intuitively, $L_1(v)$ is the vertex with lowest DFS index which is either equal to v or can be reached from v along a directed path of tree edges followed by at most one back edge. We may easily adapt Algorithm 1.1 to compute the $L_1(v)$ for all vertices $v \in V(D)$. Note that, by definition, $L_1(v) \leq dfi(v)$. Each time we visit a vertex v for the first time, we must set $L_1(v) = dfi(v)$. Each time we backtrack to the parent or scan a new edge which is a back edge, we must check whether the first lowpoint $L_1(v)$ must be updated. Thus, suppose we are scanning a back edge $v \xrightarrow{e} u$, then

if $L_1(v) > dfi(u)$ then we assign $L_1(v) = dfi(u)$.

If we are backtracking to the parent of v, namely Parent(v), then we must update $L_1(Parent(v))$ according to $L_1(v)$;

if $L_1(v) < L_1(Parent(v))$, then $L_1(Parent(v)) = L_1(v)$.

These modifications are simple to implement, and consist of the insertion of a few statements into the DFS procedure's code. The complexity of the DFS is unchanged, and is thus still linear.

The importance of the lowpoint function lies in the following theorem (see for example, [Eve79]).

Theorem 1.4: Let G be a connected graph, and let T be a DFS tree of G. A vertex $v \in V(T)$, is a cut-vertex of G if and only if

(a) $v \neq Root(T)$, has a child u in T such that $L_1(u) \geq dfi(v)$; or

(b) $v = Root(T)$, and v has at least two descendants.

Thus, by proceeding with a standard DFS, as modified in the above description, we may, using Theorem 1.4, detect the cut-vertices of G. This result is, in turn, important, because it allows one to find the blocks of a graph (see [CO90]). This fact, in conjunction with the following well-known result (see for example, [CO90]) justifies why planarity testing algorithms can be restricted to 2-connected (p, q) graphs with $q \leq 3p - 6$.

Theorem 1.5: A graph G is planar if and only if each block of G is planar.

Section 1.4

The Rotational Embedding Scheme

We follow Behzad, Chartrand and Lesniak-Foster [BCL79] for some details. A *compact orientable 2-manifold* is a surface which has a number of holes or, equivalently, handles placed on it. For the remainder of the thesis, if we refer to a surface, then we mean a compact orientable 2-manifold. The *genus* of a surface is the number of handles or holes on the surface. For a graph G, the genus $\gamma(G)$ of G is the minimum genus amongst all the genera of the surfaces on which G can be embedded. The Kuratowski graphs K_5 and $K_{3,3}$ both have genus 1. Consider Figure 1.1, below, which shows K_5 embedded on surfaces of genus 1 and 2.

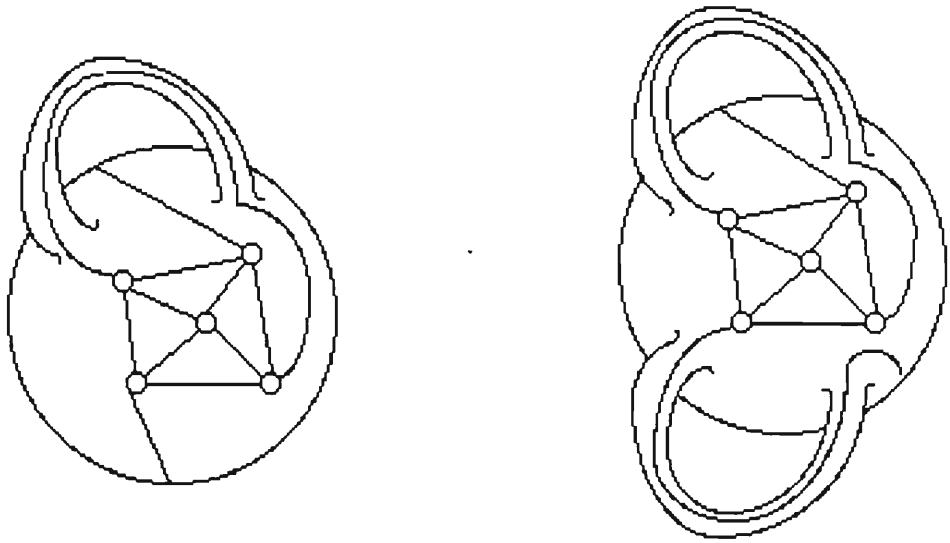


Figure 1.1 - Embeddings of K_5

A region is called a *2-cell* if any simple closed curve in that region may be continuously contracted in that region to a single point. Consider, as an example, Figure 1.2. The region R_1 is not a 2-cell, for if we place a closed curve in R_1 around R_2, then this curve cannot be contracted to a single point in R_1 (since R_2 is in the way). However, R_2 is a 2-cell.

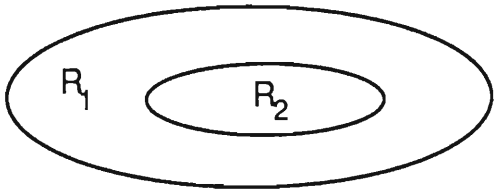


Figure 1.2 - 2-cell regions

An embedding of a graph G, on a surface S, is called a *2-cell embedding* of G on S, if all the regions of the embedding are 2-cell. We have the following extension to Theorem 1.1.

Theorem 1.6: Let G be a connected (p, q) graph, which has been 2-cell embedded on a surface of genus n, so that r regions result. Then,

$$p - q + r = 2 - 2n$$

The next result shows that every connected graph can be 2-cell embedded on at least one surface.

Theorem 1.7: If G is a connected graph, embedded on the surface of genus $\gamma(G)$, then every region of G is a 2-cell.

From two preceding results, we have the following theorem.

Theorem 1.8: Let G be a connected (p, q) graph, embedded on a surface of genus $\gamma(G)$, so that r regions are produced. Then,

$$p - q + r = 2 - \gamma(G)$$

From Theorem 1.8 we may deduce that every two embeddings of a connected graph G on the surface of genus $\gamma(G)$ have the same number of regions. The next theorem gives a lower bound for the genus $\gamma(G)$ of a graph G.

Theorem 1.9: If G is a connected (p, q) graph, then

$$\gamma(G) \geq \frac{q}{6} - \frac{p}{2} + 1$$

The next result (by Battle, Harary, Kodama and Youngs [BHKY62]) shows that, in order to be able to determine the genus of a graph it suffices to be able to determine the genera of its blocks.

Theorem 1.10: If G is a graph with blocks $B_1, B_2, ..., B_m$, then

$$\gamma(G) = \sum_{i=1}^{m} \gamma(B_i).$$

We have the following result, by Ringel and Youngs [RY68], on a formula of the genus of a complete graph

Theorem 1.11: If K_p is the complete graph on p vertices, then

$$\gamma(K_p) = \left\lceil \frac{(p-3)(p-4)}{12} \right\rceil, \qquad p \geq 3$$

The next result, by Ringel [Rin65], gives a formula for the genus of the complete bipartite graph.

Theorem 1.12: If $K_{m,n}$ is the complete bipartite graph, then

$$\gamma(K_{m,n}) = \left\lceil \frac{(m-2)(n-2)}{4} \right\rceil, \qquad m, n \geq 2$$

We may now turn our attention to describing an embedding on a surface. Consider the plane graph shown in Figure 1.3, below.

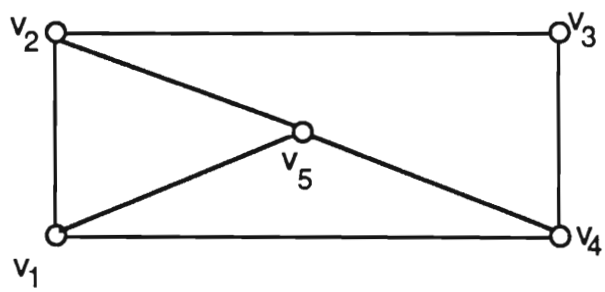


Figure 1.3 - A plane graph G

We next discuss the Rotational Embedding Scheme. This method of describing a 2-cell embedding of a connected graph on some compact orientable 2-manifold, simply visits, for each vertex v, a cyclic permutation of the edges incident with v, as they appear about v as one proceeds in anticlockwise order about v. Since every edge can be expressed as vw, we merely list w to represent this edge.

In our example, the edges around v_1 are arranged cyclically in anticlockwise order, v_4, v_5, v_2. An equivalent ordering is v_2, v_4, v_5. Thus, the ordering may be thought of as a cyclic permutation π_1, since the order is independent of the exact starting vertex adjacent to v_1. If we replace a vertex v_i with i, then $\pi_1 = \{4, 5, 2\}$. For the rest of the vertices, we obtain the following cyclic permutations.

$$\pi_2 = \{1, 5, 3\}$$
$$\pi_3 = \{2, 4\}$$
$$\pi_4 = \{3, 5, 1\}$$
$$\pi_5 = \{4, 2, 1\}$$

If we proceed around a region, tracing out the edges of the boundary of the region in a clockwise manner (that is, the boundary is on our left), we may describe a region. Consider, as an example, the region traced out if we start with the edge v_1v_5. Proceeding in a clockwise manner, around the region, we obtain the next edge v_5v_4, and hence the next (and last) edge is v_4v_1. Thus, we may describe the region by the order of vertices as they appear around the region as we proceed in a clockwise manner around the boundary of the region. We may also trace out the same region using the cyclic permutations. All we do is start with the element 5 in π_1. We choose the next element from π_5 by taking the element following element 1 in π_5, that is, the element given by $\pi_5(1) = 4$. Then, we choose the element in π_4 specified by $\pi_4(5) = 1$. Since $\pi_1(4) = 5$, we have returned to our initial edge, and so we have described a region. Each of these tracings by using the cyclic permutations are permutation cycles, called *orbits*. The regions are completely described by all possible orbits on the cyclic permutations. Thus, we obtain the following four regions.

$R_1 : v_1 - v_5 - v_4 - v_1$

$R_2 : v_1 - v_2 - v_5 - v_1$
$R_3 : v_1 - v_4 - v_3 - v_2 - v_1$
$R_4 : v_2 - v_3 - v_4 - v_5 - v_2$

Thus, with the aid of the Rotational Embedding Scheme we can describe the regions of an embedding of G on some surface. If a graph G has vertex set V(G), then, for a vertex $v_i \in V(G)$, V(i) denotes all vertices adjacent to v_i. We have the following theorem (due, in its present form, to Youngs [You63]).

Theorem 1.13: Let G be a connected graph, with $V(G) = \{v_1, v_2, ..., v_p\}$. For each 2-cell embedding of G on a surface, there exists a p-tuple $(\pi_1, \pi_2, ..., \pi_p)$ where, for i = 1, 2, ..., p, $\pi_i : V(i) \rightarrow V(i)$ is a cyclic permutation which describes the subscripts of vertices adjacent to v_i as they appear in anticlockwise order around v_i.
Conversely, for each such p-tuple $(\pi_1, \pi_2, ..., \pi_p)$, there exists a 2-cell embedding of a graph G on some surface such that for i = 1, 2, ..., p, the subscripts of the vertices adjacent to v_i, and in anticlockwise order around v_i, are given by π_i.

Thus, it is sufficient to describe a graph using the Rotational Embedding Scheme. In general, if a graph is planar, then there may be several p-tuples which describe embeddings of the graph in the plane. However, in some instances, a planar graph may have a unique embedding in the plane. We have the following characterisation (for example [Wil85]).

Theorem 1.14: Let G be a planar, graph. Then, G has a unique embedding in the plane if and only if G is 3-connected.

We close this section with a few well-known results on the maximum genus of a graph.

For a graph G, the *maximum genus* $\gamma_m(G)$ of G is the maximum genus among all the genera of the surfaces on which G can be 2-cell embedded. We have the following result.

Theorem 1.15: A connected graph G has a 2-cell embedding on a surface S_k if and only if

$$\gamma(G) \leq k \leq \gamma_m(G)$$

The final results provide upper bounds on the maximum genus $\gamma_m(G)$ of a graph G.

The next two results on the maximum genus are due to Nordhaus, Stewart and White [NSW71].

Theorem 1.16: Let G be a connected graph, then

$$\gamma_m(G) \leq \left\lfloor \frac{q - p + 1}{2} \right\rfloor$$

Theorem 1.17: If K_p is the complete graph on p vertices, then

$$\gamma_m(K_p) = \left\lfloor \frac{(p - 1)(p - 2)}{4} \right\rfloor, \qquad p \geq 3$$

Finally, we have the following result on the maximum genus of a complete bipartite graph by Ringeisen [Rin72].

Theorem 1.18: If $K_{\ell,n}$ is the complete bipartite graph, then

$$\gamma_m(K_{\ell,n}) = \left\lfloor \frac{(\ell - 1)(n - 1)}{2} \right\rfloor, \qquad m, n \geq 2$$

Section 1.5

Graph Data Structures

In this section we look at various techniques for storing a graph and its edges, as well as some of the more common data structures which we use for the rest of the chapters. We base some of the results of this section on the results by Sutcliffe [Sut85]. Sutcliffe did a comprehensive study and analysis of various representations, including the three most common representations, namely adjacency matrices, adjacency lists and incidence matrices. The main purpose is to familiarise the reader with some of the fundamental operations which are possible with the data structures that we use in this thesis. A good introduction into linked lists was done by Horowitz and Sahni [HS76].

The main prerequisites of the data structure we must select are:

(a) Provision must be made for the capability of multiple attributes for the vertices and edges (i.e. we must be able to associate a variety of quantities with the vertices and edges).
(b) Space requirements must be linear in the number of vertices and edges. An algorithm which has time complexity O(p) but uses $O(p^2)$ space is of reduced value.
(c) The representation must allow for the easy removal and addition of vertices and edges. During some of the algorithms we present in later chapters, we sometimes need to perform removals and additions.
(d) There must be a facility to allow for the reordering of the edges in the data structure. If we are to obtain an embedding, then there must be a simple technique to allow the sorting of the edges in the representation, so that we may note the order in which they appear in the embedding.

One of the more common representations is the adjacency matrix representation. Here, we represent the graph by a p x p boolean matrix A, where A(i, j) is true if and only if there is an edge from vertex i to vertex j. The representation has a certain amount of flexibility, in that it allows for directed graphs, and has an easily understood representation. The large drawback is that it uses $O(p^2)$ space. There is also no direct method of ordering the edges incident with a vertex. Note that the addition of extra vertices during the running of the algorithm must be anticipated before the start of the algorithm.

The next representation we shall discuss is the incidence matrix representation. Here, the graph G is represented by a p x q matrix B, where B(i, j) = k, if vertex i is adjacent to vertex k along edge e(j) (that is, $e(j) = v_i v_k$). The incidence matrix representation is an unusual representation, and has the disadvantage of having to bound the number of edges in G. Thus, the addition of any extra edges during the execution of the algorithm is not easy. The space required is O(p * q), and thus is not favourable. Lastly, there is no easy method of representing the order in which the edges appear around a vertex in an embedding.

Both of the above representations have the draw back of the excessive space wastage, and of the lack of flexibility in representing an embedding of a graph. The closest one might come to representing an embedding using either of the two representations is to assign labels to the edges appearing around a vertex v, where the label for an edge e specifies the position in the sequence of edges around v, in the embedding that the edge e appears. Possibilities then exist for simulating some form of linked list, by pointing to the array position where the next edge appears. This solution, however, is artificial. The next representation lends itself naturally to the solution of the above representation problems.

The standard adjacency list representation represents a graph G by a vector of p headers and a set of p linked lists. Each header represents a vertex and contains a pointer to the corresponding linked list, whilst the corresponding linked list represents edges incident to that vertex. One disadvantage of the representation is that p must be known beforehand.

But, we note that all of the algorithms covered in the following chapters are well behaved, and never require space for more than one extra vertex. Another disadvantage is that there is very little clarity when working with linked lists, as compared to a representation like an adjacency matrix. We feel that the flexibility of the representation more than compensates for the above disadvantage. As we shall show during the rest of this section, the flexibility of the representation is extensive.

Sutcliffe [Sut85] concurs with the above conclusion noting that, in general, the linked list representations of graphs are the most flexible, are space efficient, and outperform the other representations in terms of graph operations. The one serious drawback with the representation, which was noted by Sutcliffe, was that it does not lend itself to an efficient check for "is vertex x adjacent to vertex y". However, this check is not needed during the running of the algorithms covered in this thesis.

The adjacency list representation is the representation which we use throughout this thesis. We call the linked list corresponding to vertex v, the *adjacency list of v.* In the vector of p headers we store a pointer element, which points to the start or head of the adjacency list. The tail of the adjacency list is the very last element in the adjacency list. Figure 1.4, below, shows a typical linked list representation for a vertex v. Note the symbol for the end of a list, called the *nil pointer.*

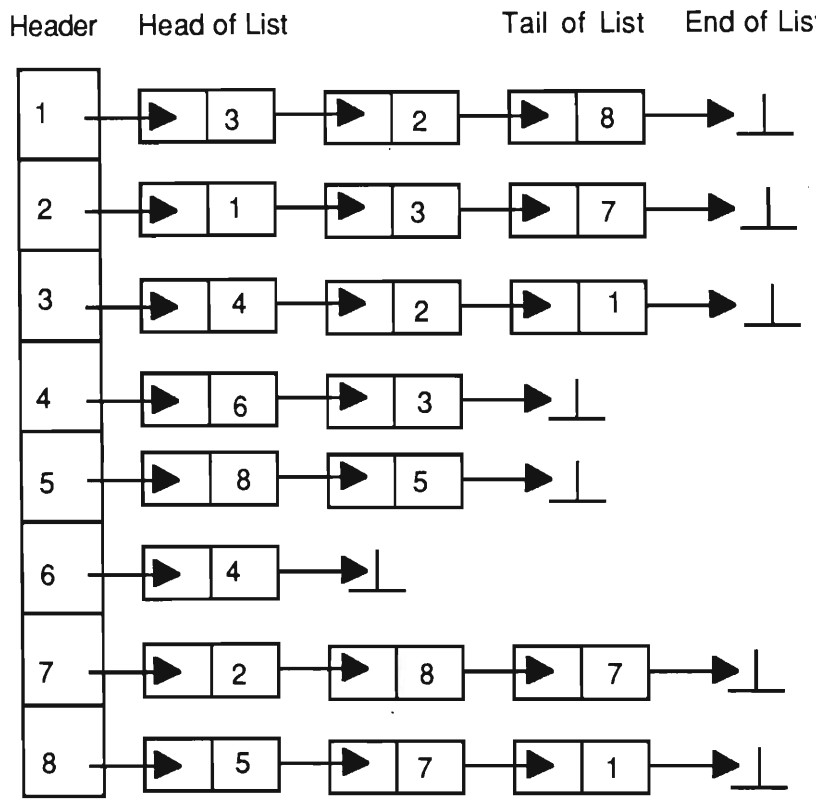


Figure 1.4 - A adjacency list Representation for a graph G

Thus, from Figure 1.4, we may observe that, for example, vertex 6 is only adjacent to vertex 4, and that vertex 1 is adjacent to vertices 3, 2 and 8.

When adding an edge element into an adjacency list, we merely swop a few pointers. We make the new edge element point to the head of the list, and make the header point to the new element. Figure 1.5, below, illustrates this idea.

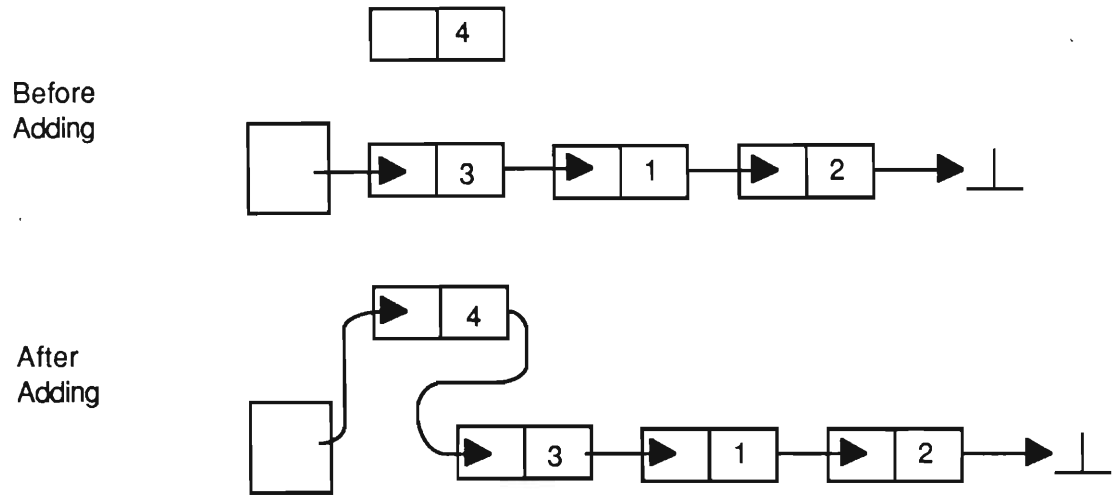


Figure 1.5 - Adding to an adjacency list

A modification of the adjacency list structure which we sometimes use is for the linked list to be changed into a doubly linked circular list. Notice from Figure 1.4, that each edge in the adjacency list of a vertex only knew what its successor was in the list. There is no mechanism for an edge element to see what the previous edge element was. Thus, deletion from the list is tedious, since we have to scan from the start of the list, looking for the correct edge to delete, as well as noting what the previous edge in the adjacency list was. A doubly linked adjacency list is one where each edge knows the previous edge and the next edge in the adjacency list. A doubly linked circular adjacency list is one where we do not explicitly note the start and end of the list. Figure 1.6, below, shows the two types of linked lists.

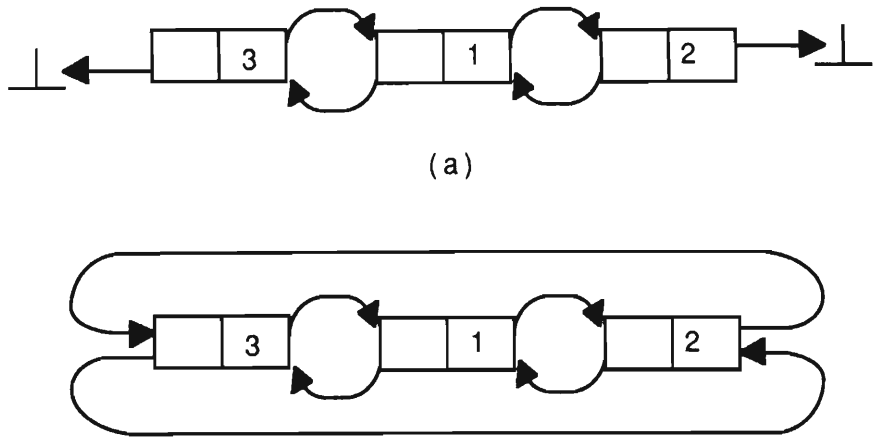


Figure 1.6 - (a) A Doubly Linked adjacency list; (b) A Doubly Linked Circular adjacency list

Note that we also use the doubly linked lists for other operations, therefore, for the rest of this discussion, we shall refer solely to the lists themselves, rather than to the adjacency lists. The advantage of a doubly linked list is that the deletion of an edge is very easy. This is even more true for the case of a doubly linked circular list. Figure 1.7, below, shows an example of the deletion of an element from a doubly linked list.

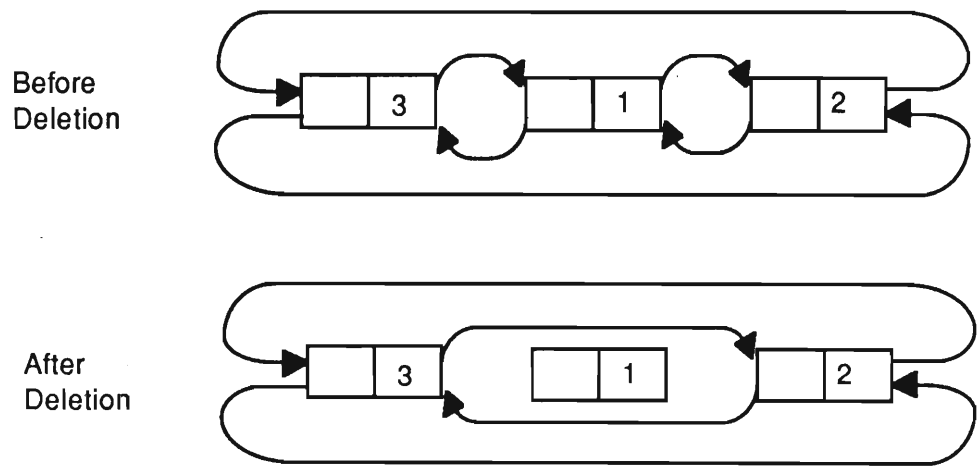


Figure 1.7 - Deleting an element from a Doubly Linked List

The circular linked list representation is very flexible when we wish to merge two doubly linked lists together. Figure 1.8, below, gives an

example of merging two circular lists A, and B, to produce a single list A. Notice how the only work performed is by changing the values of four pointers, namely the start and end pointers of each list.

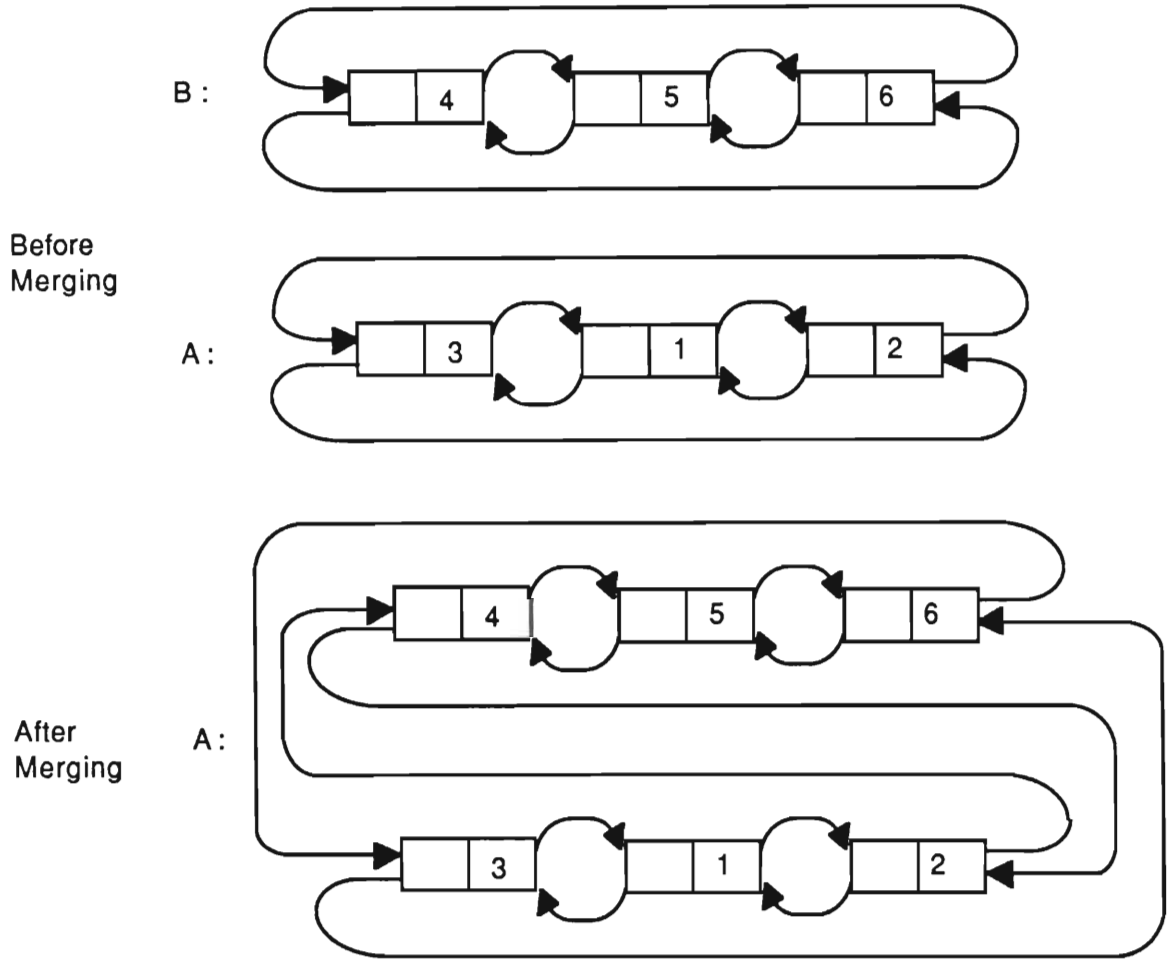


Figure 1.8 - Merging two circular lists

The last observation that we make is that, for adjacency lists, it is sometimes convenient for an edge to know the location of its counterpart in the other adjacency list. That is, if a vertex v is adjacent to a vertex w, then the reference in the adjacency list of v to w also keeps a store of the reference in w's adjacency list to v. This extra field of information is extremely convenient when, for example, we wish to efficiently delete the edge vw. If we only know the location of the reference in the adjacency list of v to w, then by referencing this extra field, we may avoid scanning through the adjacency list of w to look for the reference to v.

Chapter 2

Planarity Testing Algorithms

In this Chapter we introduce three planarity testing algorithms. They all use different approaches for testing whether or not a graph is planar.

The first algorithm we discuss is a well known algorithm by Demoucron, Malgrange and Pertuiset [DMP64]. This algorithm is efficient, but certainly not linear. It has a naive implementation complexity of $O(p^4)$, where p is the order of the graph, and uses *path addition*. By path addition we mean that the algorithm adds one path at a time to the plane subgraph it has built thus far. We also show that, through careful programming, we may reduce the complexity of the algorithm to $O(p^3)$.

The last two algorithms we discuss are both linear algorithms in the number of edges or vertices, that is their complexities are O(max (p,q)), where p and q denote the order and size of the graph. They are the only two linear planarity testing algorithms known. The first linear algorithm was suggested by Auslander and Parter [AP61] and Goldstein [Gol63], but the current algorithm, from a complexity and graph theoretic view, is due to Hopcroft and Tarjan in 1974 [HT74]. They use the concept of *fragment addition* to derive a recursive fragment addition algorithm, where an entire fragment is (recursively) added to the planar subgraph built.

The last of the two linear algorithms is due to Lempel, Even and Cederbaum [LEC67]. Their original algorithm was not linear, but Even and Tarjan [ET76] showed how the first part of the algorithm can be implemented in linear time and Booth and Lueker in 1976 [BL76] showed how, through the use of the PQ-tree data structure, the latter part of the algorithm may be implemented in linear time. The algorithm is the most complex of the three algorithms studied, but on the other hand has the advantage that it lends itself to a linear embedding algorithm (Chiba,

Nishizeki, Abe and Ozawa [CNAO85]). The algorithm is based on a *vertex addition* concept, whereby a vertex at a time is added to the plane subgraph built thus far.

Section 2.1

The Demoucron, Malgrange and Pertuiset Path Addition Algorithm

We use the description by Chartrand and Oellermann [CO90] as a guide. Let G be a 2-connected (p, q) graph with $q \leq 3p - 6$. Let H be a planar subgraph of G. Suppose that in an embedding of H in the plane there are k regions, $R_1, R_2, \ldots, R_k$. If a fragment $\mathcal{F}_i$ of G with respect to H has all its attachments in R, then we say that $\mathcal{F}_i$ is an *$\mathcal{R}$-fragment* of H. The set of all regions for which $\mathcal{F}_i$ is an R-fragment is denoted by $\mathcal{R}(\mathcal{F}_i, H)$. For example, Figure 2.1 shows a graph G, a plane subgraph H of G, and the fragments $\mathcal{F}_i$ of G with respect to H. From Figure 2.1, $\mathcal{R}(\mathcal{F}_1, H) = \{R_1\}$, $\mathcal{R}(\mathcal{F}_2, H) = \{R_1, R_2\}$ and $\mathcal{R}(\mathcal{F}_3, H) = \{R_1, R_3\}$.

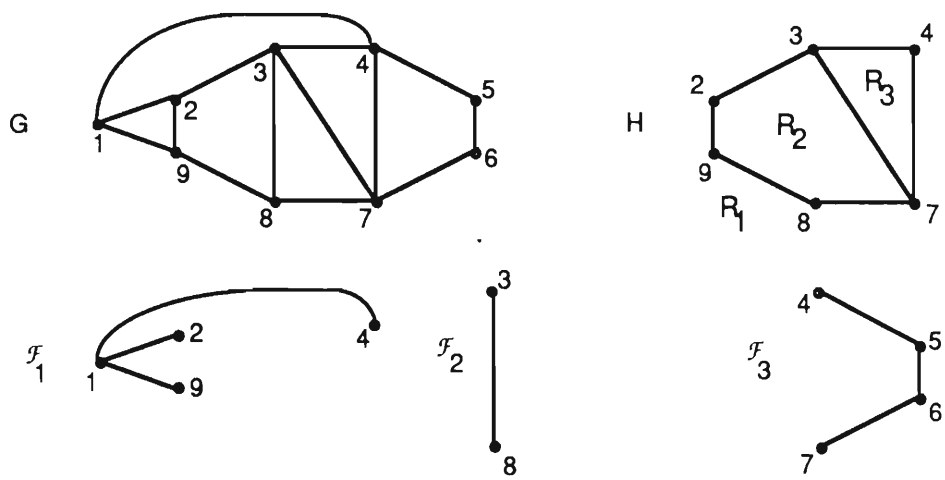


Figure 2.1 : Graph G, a plane subgraph H of G and Fragments $\mathcal{F}_i$ of H.

The Demoucron, Malgrange and Pertuiset Algorithm begins by finding a cycle $H = G_1$ in G and embeds it in the plane. The exterior region is called R_1 and the interior region R_2, say. In general, suppose G_j has been defined. Then all fragments $\mathcal{F}_i$ with respect to G_j are determined. For every fragment $\mathcal{F}$ of G_j we determine $\mathcal{R}(\mathcal{F}, G_j)$. If $|\mathcal{R}(\mathcal{F}, G_j)| = \emptyset$ for some fragment $\mathcal{F}$, then G is non-planar and we stop. If $|\mathcal{R}(\mathcal{F}, G_j)| \geq 1$ for all fragments but if there exists a fragment $\mathcal{F}$ such that $|\mathcal{R}(\mathcal{F}, G_j)| =$

1 then we consider such a fragment. Otherwise, $|\mathcal{R}(\mathcal{F}, G_j)| \geq 2$ for all fragments and we select a fragment $\mathcal{F}$ at random. Since G is 2-connected, we may choose, from the selected fragment $\mathcal{F}$, a non-trivial path P starting and ending with vertices of G_j. This path we embed in a region of $\mathcal{R}(\mathcal{F}, G_j)$. By embedding the path, we split one of the regions of G_j into two regions and thereby introduce a new region. Now let G_{j+1} be the plane graph obtained by embedding P in a region of G_j. Replace j with j+1, and repeat the above process until all vertices and edges of G have been added to G_j (i.e. H is isomorphic to G) and G is planar, or until for some j there exists a fragment $\mathcal{F}$ of G_j for which $|\mathcal{R}(\mathcal{F}, G_j)| = \emptyset$ in which case G is non-planar.

Before we describe the Demoucron, Malgrange and Pertuiset Algorithm, we present a few useful algorithms. Firstly, we need to discuss the problem of finding an initial cycle to be the initial graph G_1 in our algorithm. To find such a cycle, we merely perform a Depth-First Search (DFS) on G, together with the generation of the first lowpoint function $L_1(v)$. Recall from Chapter 1 that the first lowpoint function $L_1(v)$ indicates the lowest labelled vertex which can be reached from v in the DFS tree T via a series of tree edges followed by at most one back edge. Also, the DFS together with the generation of $L_1(v)$ values may be performed in O(q) steps. Suppose we have DFS tree T, rooted at u. Then, since we are only considering 2-connected graphs, they contain cycles. For example, the first time we encounter a back edge from a vertex v to u, the u - v path in T, as well as the back edge $v \rightarrow u$ forms a cycle.

To find a cycle, we perform a DFS on G, building a DFS tree T with root u. We construct a path P. Initially P is empty, and we set w = u. Then, we choose the edge e = wv, incident with w, such that $L_1(v) = 1$. We add the edge to P. Now, let w = v. Choose an edge wx incident with w such that $L_1(x) = 1$, and add wx to P. We repeat this process, proceeding down the DFS tree, until we encounter a back edge wu, incident with our root vertex u. The required cycle is given by P + {wu}.

The next tool we require for the Demoucron, Malgrange and Pertuiset Algorithm is an efficient procedure, essentially a DFS, for determining

the fragments of G with respect to a subgraph H. For each fragment $\mathcal{F}$ we construct a spanning DFS tree $T_{\mathcal{F}}$, and we denote by Root($\mathcal{F}$) the root of $T_{\mathcal{F}}$.

Initially all edges in H are marked scanned, and all other edges in G - H are marked unscanned. The main loop of the algorithm runs through each vertex v_i of H. If there is an edge $e = v_iu$ incident with v_i, which has not been scanned, then we start constructing a new fragment, $\mathcal{F}$ say. If no such edge exists, then we have considered all incident edges of v_i and we consider a new vertex of H, if such a vertex remains. Otherwise we have found all the fragments, and halt. Let us consider the first case where we are starting a new fragment $\mathcal{F}$. Root($\mathcal{F}$) is set to v_i, we add $e = v_iu$ to $\mathcal{F}$ and mark e as scanned. We use a variable w to denote the current active vertex of the DFS whilst determining the current fragment. Initially, w is set equal to u and Parent(w) is set equal to v_i. We then proceed to build the rest of the fragment. Algorithm 2.1a, below, shows the algorithm so far.

Algorithm 2.1a: Generate_Fragments (G, H)

```
        { Generate the fragments of G with respect to H }

        variable Scan(e) is a boolean array of q elements

        For each edge e ∈ E(G)              { initialise }
            if e ∈ E(H)
                then Scan(e) = true
                else Scan(e) = false

        While vertex v ∈ V(H) is unvisited do
            Parent(v) = ∅          { root of a tree T_F has no parent }
            For each edge e = vu incident with v do
                if not Scan(e)
                    then           { we have found a new fragment F }
                        Root(F) = v
                        Scan(e) = true
                        F = F + e  { add first edge to fragment tree }
                        w = u
                        Parent(w) = v
                        Build_Fragment (F, w)
                        Output F
end
```

In Build_Fragment we enter a new loop which ends only when we arrive back at Root($\mathcal{F}$) (i.e. when w = Root($\mathcal{F}$)). There are two options. Either w $\in$ V(H) or w $\notin$ V(H). For the first case, w is an attachment of $\mathcal{F}$, we note it as such, we set w to Parent(w) and loop. If w $\notin$ V(H), then we scan edges of the type e = wv. If the vertex v is already in $\mathcal{F}$, then we mark e as having being scanned and we consider the next edge. If v is not in $\mathcal{F}$, then we add e = wv to $\mathcal{F}$, mark e as having been scanned, set Parent(v) = w, set w to v and loop. If all edges incident with w have been scanned, then we backtrack, i.e. set w to Parent(w) and loop.

This straightforward algorithm generates as output for each fragment $\mathcal{F}_j$ a spanning tree of $\mathcal{F}_j$, called the *Fragment Tree* of $\mathcal{F}_j$. Note that this tree is sufficient for our purposes since we only need a path between two attachments of $\mathcal{F}_j$. We detail the complete algorithm for building a fragment below.

Algorithm 2.1b: Build_Fragment ($\mathcal{F}$, w)

```
        { Build a fragment F with root w }

        variables
            We use the same array Scan(e) as in the calling Algorithm 2.1a

        repeat
            if w ∈ V(H)
                then
                    Note w is a new attachment of F
                    w = Parent (w)              { backtrack to parent in tree }
            else
                if there is an edge e = wv such that Scan (e) = false
                    then                        { found an unscanned edge of F }
                        Scan(e) = true
                        if v ∉ V(F)
                            then                { found an unexplored vertex of F }
                                F = F + e       { add edge to tree }
                                Parent(v) = w
                                w = v           { and explore }
                            else
                        else                    { no unscanned edges }
                            w = Parent (w)      { so backtrack to parent }
        until w = Root (F)
end
```

As an example of the fragment generation process of this algorithm, consider the graph in Figure 2.2, below, and its associated fragment trees.

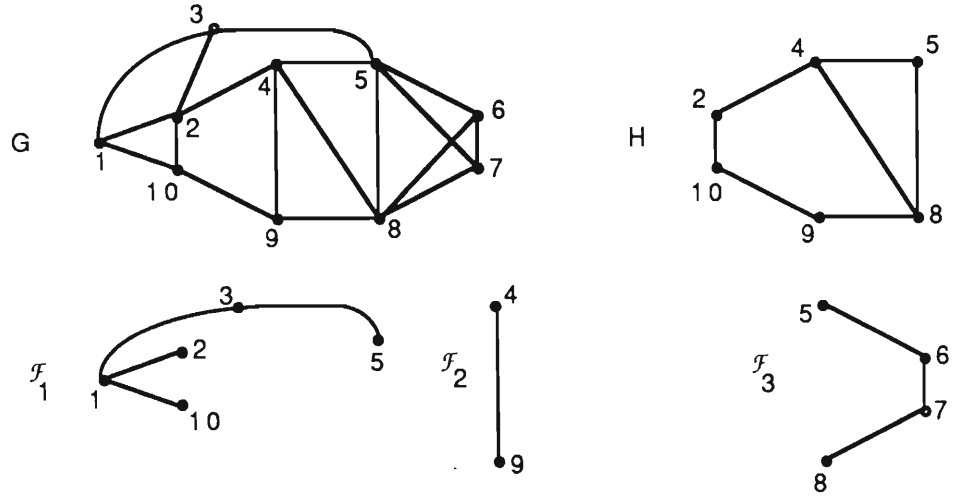


Figure 2.2 : A graph G, plane subgraph H, and associated Fragment Trees

We now analyse the complexity of Algorithm 2.1(a and b):

Lemma 2.1: Algorithm 2.1 may be performed in O(p) steps.
Proof: Consider first the initialisation stage of the algorithm. The edge scan values are assigned in O(q) steps. Since $q \leq 3p - 6$, the initialisation phase has complexity O(p). In the main algorithm, it is easy to verify that once an edge e is scanned (i.e. Scan(e) is set to true), we never traverse that edge again. Furthermore, we traverse only those edges which are not already in H, and those edges are immediately noted as scanned. Thus each edge is traversed at most once, and thus the main algorithm has complexity O(p). Hence the entire algorithm has complexity O(p), and the lemma is proved. ❑

Before we proceed, we shall need some extra definitions. Consider a planar subgraph G_i of a planar graph G. We say that a plane realisation $\hat{G}_i$ of G_i is G-*extendible* if there exists a plane realisation $\hat{G}$ of G which has $\hat{G}_i$ as a plane subgraph, i.e. $\hat{G}_i$ may be extended to $\hat{G}$. It may happen that

a plane subgraph of G is not G-extendible. See Figure 2.3 which shows G $\cong K_5 - e$, where e is an edge of K_5, and two plane realisations $\hat{H}_1$ and $\hat{H}_2$ of the same subgraph H.

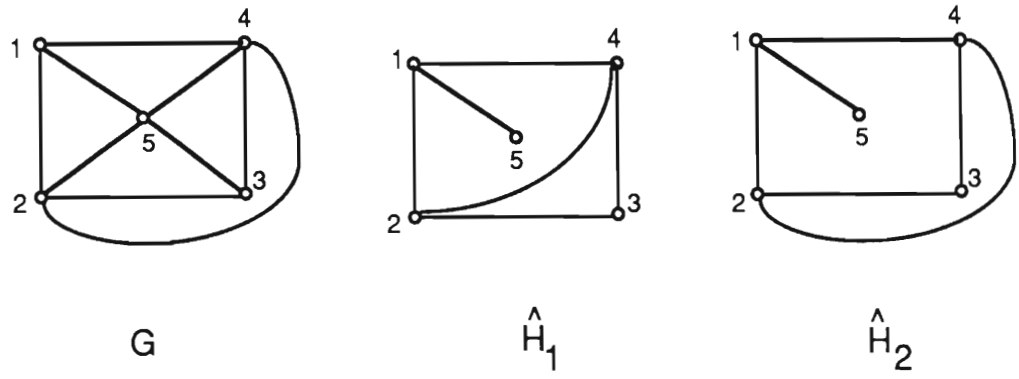


Figure 2.3 : K_5-e and two plane subgraphs $\hat{H}_1$ and $\hat{H}_2$

Note that G is planar, and that $\hat{H}_2$ is G-extendible. However, $\hat{H}_1$ is not G-extendible since v_5 is in the interior of the triangle bounded by v_1, v_2, v_4, v_1. Thus, it is impossible to add the edge v_3v_5 to $\hat{H}_1$ without crossing other edges.

We can now present the algorithm by Demoucron, Malgrange and Pertuiset:

Algorithm 2.2: Demoucron_Malgrange_Pertuiset

{ Test whether a 2-connected graph with $q \leq 3p - 6$ is planar }

```
Depth_first_Search (G)
Find_Cycle (G1, G)

    { Suppose G1 is the cycle v1, v2, ..., vk, v1}

Add_to_Region (1, v1, ..., vk)
Add_to_Region (2, v1, ..., vk)
i = 1
while |E(G)| ≠ |E(Gi)| do
    Generate_Fragments (G, Gi, F1, ..., Fn)
    For Loop = 1 to n do
        Find_Common_Regions (FLoop, R( FLoop, Gi))
```

```
        if | ℛ( ℱLoop, Gi)| = 0
            then
                write ('The graph is not planar')
                Halt
    If | ℛ( ℱj, Gi)| = 1 for some fragment ℱj
        then
            ℱ = ℱj
            R is the only element of ℛ( ℱj, Gi)
        else
            Choose a fragment ℱj at random
            ℱ = ℱj
            R is any Rk ∈ ℛ( ℱ, H)
    Determine_Path_in_F ( ℱ, P:x1, x2, ..., xk)
    V(H) = V(H) ∪ V(P)
    E(H) = E(H) ∪ E(P)

    { We now assume that vertices bounding region R are v1, ..., vs, vs+1,
...., vm, vm+1, ..., vr, where vs = x1 and vm = xk }
    { see Figure 2.4, below }

    Change_Region (R, v1, ..., vs-1, x1, x2, ..., xk, vm+1, ..., vr)
    Add_New_Region (Ri+2, vs, vs+1, ..., vm-1, xk, xk-1, ..., x2)
    i = i + 1
```

end

The algorithm is straightforward. To generate the fragments for G_i, we use Algorithm 2.1. Note that determining a path in a fragment $\mathcal{F}$ can be accomplished by the Parent(v) values calculated in Algorithm 2.1. Splitting a region into two new regions by the addition of a path is shown below in Figure 2.4.

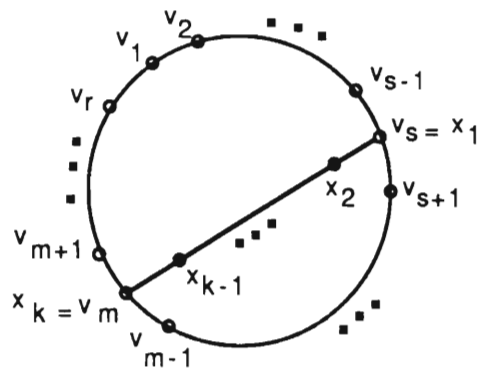


Figure 2.4 : Splitting a region into two regions by a new path

If we use a linked list data structure to store the vertices which bound a region, then the splitting of the region into two regions is easily accomplished by a traversal of the list of the original region. We defer further discussion of this operation until the complexity discussion of Algorithm 2.2, below.

Theorem 2.1: Algorithm 2.2 correctly tests for planarity of a 2-connected graph G.

Proof: We proceed by induction on i to show that, if G is planar, then G_i is G-extendible for all i ($1 \leq i \leq n$), where n is the number of iterations of the main loop. Suppose G is planar. Clearly, G_1 is a cycle, and as such must be G-extendible since the cycle must exist in any plane realisation of G. Now we assume that G_i is G-extendible. Consider the fragment addition process. Since G_i is G-extendible, $|\mathcal{R}(\mathcal{F}_j, G_i)| \geq 1$ for all fragments $\mathcal{F}_j$ of G_i. We consider two cases.

Suppose we reach the stage in Algorithm 2.2 where we have chosen a fragment $\mathcal{F}$ to be embedded in a region R. If $|\mathcal{R}(\mathcal{F}, G_i)| = 1$ then $\mathcal{F}$ must be embedded in R in an extension of G_i to a plane embedding of G, so G_{i+1} is G-extendible.

Consider the latter case, where $|\mathcal{R}(\mathcal{F}, G_i)| > 1$ for all fragments $\mathcal{F}$ of G_i. We need to show that if we choose $\mathcal{F}$ and embed it in R, then the resulting graph G_{i+1} is G-extendible. We note that the only fragments which are affected by the embedding of $\mathcal{F}$ in R are those which interlace with $\mathcal{F}$. Now, suppose G_i may be extended to a planar realisation $\hat{G}$ of G and that $\mathcal{F}$ is embedded in a region R' of $\hat{G}$. If R' = R, then G_{i+1} is G-extendable. Otherwise, if R' ≠ R, let $\mathcal{F}(R, R')$ denote the set of all of the fragments which are both R- and R'-fragments. Since $|\mathcal{R}(\mathcal{F}, G_i)| > 1$ for every fragment $\mathcal{F}$ of G_i, we may move all fragments $\mathcal{F}' \in \mathcal{F}(R, R')$ embedded in R to the region R' and all those fragments $\mathcal{F}' \in \mathcal{F}(R, R')$ embedded in R' to R across the common boundary. The resulting planar realisation $\hat{G}'$ of G has $\mathcal{F}$ embedded in region R. Thus, if in G_i we embed $\mathcal{F}$ in R, we must have that G_{i+1} is also G-extendible. ❑

To conclude this section we derive an expression for the overall complexity of Algorithm 2.2, and give a more efficient implementation of Algorithm 2.2.

Theorem 2.2: The complexity of Algorithm 2.2 is $O(p^4)$.

Proof: As discussed in Chapter 1, the DFS has complexity O(p). Consider the cycle generation. The worst case for determining a cycle in a 2-connected (p, q) graph, is when the graph is itself a cycle. So this step has complexity O(q) = O(p). The main while statement, namely, while $|E(G)| \neq |E(G_i)|$ do, is performed at most O(q) times, once for each edge not in the original cycle. We have already seen that Algorithm 2.1 has complexity O(p) steps. Note also that if we add a step to Algorithm 2.1 which updates a linked list of the attachments, of each fragment, the complexity of Algorithm 2.1 is unaltered. For every fragment $\mathcal{F}_j$ we keep Root($\mathcal{F}_j$) at the head of this list, and if v is an attachment, we store Parent(v) for $\mathcal{F}_j$ as well.

To discuss the complexity of the Find_Common_Regions procedure call and Change_Region and Add_Region procedure calls, we need to introduce some data structures. With every region R_j we associate a doubly linked circular list which lists the vertices of the cycle which bound R_j, in order of occurrence on the cycle. We denote this list by *Cycle(R_j)*. Further, with every vertex we associate a linked list which stores the regions on whose boundary the vertex lies, sorted in order of increasing region number. This list is denoted by $\mathcal{R}(v, G_i)$. After the initial cycle $\hat{G}_1$ is determined we have that $\mathcal{R}(v, G_1) = \{R_1, R_2\}$, for every vertex $v \in V(\hat{G}_1)$. Every vertex $v \notin V(G_1)$ has $|\mathcal{R}(v, G_1)| = \emptyset$. Furthermore, the linked lists Cycle(R_1) and Cycle(R_2) are easily constructed as we construct the initial cycle G_1. Note Cycle(R_j) = $\emptyset$ for $j \notin 1,2$.

Now, to perform the call Find_Common_Regions for a certain fragment $\mathcal{F}_j$ of G_i, we get the first two attachments of that fragment and build a linked list of the initial list of common regions as follows. Initialise a new list, called *Common List* to nil. We compare the first two elements on the region lists of the two attachments. If the regions are equal, then we add this region as common to our Common List, and move to the next

element in each of the two lists. If the regions are not equal, then we move to the next element in the list which has the element with the smaller region number. We repeat the process until we reach the end of one of the lists. We make a maximum of i+1 comparisons since there are a maximum of i+1 regions in G_i. With this initial list we then compare every other attachment's region list against the Common List built up so far as follows. We compare the first elements of the Common List and the attachment's list. If the elements are equal, then we move to the next element in each of the two lists. If the attachment's region number is smaller, then we move to the next element in the list. Otherwise the element of the Common List is no longer common to the attachments checked so far, and we remove it from Common List. The removal is easy to perform, and so we have a total of O(i+1) comparisons for each attachment. There are at most p attachments for any fragment. Thus in total we have O((p-1) i) comparisons to generate Common List for a fragment $\mathcal{F}_j$, and so a total of O(p q i) = $O(p^2 i)$ comparisons for the for loop since we have at most q fragments.

The for loop is performed O(q) = O(p) times, and thus to find Common List for every fragment $\mathcal{F}_j$ for every $\hat{G}_i$ requires $O(p^3 * i) = O(p^4)$ steps in total during the algorithm. The checking stage where we search for some fragment $\mathcal{F}_j$ which has $|\mathcal{R}(\mathcal{F}_j, G_i)| = 1$ may be performed in order of the number of fragments, i.e in O(q) = O(p) steps. These steps have an overall complexity of $O(q^2) = O(p^2)$. The path P in the chosen fragment may be determined easily via the Parent(v) values. We merely backtrack along a path starting at the second attachment in the attachment list. Since the first attachment in the list is Root($\mathcal{F}$), the path we find will go up the fragment tree to Root($\mathcal{F}$). The path determination therefore traverses, in total during the entire algorithm, O(q) edges whilst backtracking to Root($\mathcal{F}$) for each chosen fragment $\mathcal{F}$.

Lastly, we need to examine the operations involved in embedding the new path in G_i to generate G_{i+1}. Suppose that the new path has start vertex v and end vertex w. The first operation involves the addition, for the new region R", of list of boundary cycle vertices - Cycle (R"), and the adjustment, for the old region R, of the list Cycle (R).

All we do is scan along Cycle (R) until we reach v. Then we split Cycle (R) at this point, and start Cycle (R"), continuing along the old Cycle (R) until we reach w. We then split the old Cycle (R) again at this point. The path Cycle (R") forms the start of the description of the boundary of the new region. The remainder of Cycle (R) will form part of the description of the redefined region R, and starts at the end vertex of the path to be embedded. Then, to complete the region descriptions we merely add the path to the lists made so far. The new region must add the path to Cycle (R") in reverse order, and the old region adds the path to Cycle (R) in the same order as the path is traversed. Let the vertices of the path to embed be $v = x_1, x_2, ..., x_k = w$. To add the path in reverse order we merely insert at the front of the list Cycle (R") the elements x_i $(1 \leq i \leq k)$, starting from x_1. To add the new path to Cycle (R), we simply concatenate at the end of Cycle (R) the next element x_i $(1 \leq i \leq k)$. Lastly, we adjust the links to make the lists circular.

It may be seen from the above discussion that the processing of the regions has complexity O(p). Again, we perform this task a total of O(q) = O(p) times, thus the step requires a total of $O(p^2)$ operations.

Finally, we need to maintain, for each vertex, the list of regions to which it belongs at any point in the algorithm. We need a total of O(i+1) comparisons to delete the old region and add the new region - we merely scan the linked lists and make the appropriate changes where necessary. This operation needs to be performed for all vertices which appear on the boundary of the new region which were in the old region (except the start and end vertices of the new path, to which we just add the new region using O(i+1) comparisons). We can do this O(p) times, once for each vertex in the old region, and thus, in total, we perform O(p (i+1)) operations in this pass. Overall we perform $O(q^3) = O(p^3)$ operations during the entire algorithm. All the new vertices added when we add the new path to G_i require a new list of two elements consisting of the old region and the new region. We may do this in a total of O(p) operations and in fact throughout the entire algorithm (since we only embed a path once). It may be seen from the above discussion that the algorithm may be satisfactorily executed in $O(q^4) = O(p^4)$ steps. ❑

Note that the above discussion ignores the possibility of optimising the time used by the algorithm. Although we cannot significantly reduce the worst case complexity for the algorithm, we may improve on it.

We refine the algorithm complexity by observing that it is not necessary to regenerate the fragments after every path is embedded. In practice we refine this step down to only generating the new fragments formed when embedding the path.

The only new fragments are the "splinters" of the fragment $\mathcal{F}$ which we chose to embed a path from (i.e. they have attachments on the path we just added). Thus to generate the new fragments we merely mark the vertices which are in the path we added, and look for edges not in E(H) which are incident to these vertices. This technique avoids the necessity of regenerating all fragments. However, this does not guarantee an improvement in complexity. We may further note that these new fragments only have at most two common regions. So the assignment of common regions to the new fragments is performed in O(p) time for each new fragment, thus $O(p^2)$ in total during the algorithm.

The updating of the common regions of fragments is a little more complex. Suppose that R is the region we selected to embed a fragment $\mathcal{F}$ in. Further, suppose that R" and R' are the two new regions which are produced when embedding the path in R. We choose to give R" the same region number as R. Consider any other fragment $\mathcal{F}_i$ say, that was an R-Fragment before embedding $\mathcal{F}$. Then there are four cases:

(a) $\mathcal{F}_i$ has attachments only in the redefined R. In this case we do nothing.
(b) $\mathcal{F}_i$ has attachments only in R', then we must traverse the list of regions common to all attachments of $\mathcal{F}_i$ and delete the reference to R and add R' at the tail of the list.
(c) $\mathcal{F}_i$ has only two attachments, namely the start and end vertices of the path we just added. In this case we must traverse the list and add a new region R' at the tail of the list.

(d) Lastly, $\mathcal{F}_i$ has attachments in both R and R'. In this case we remove R from the list of common regions.

It is easy whilst splitting the original region R to note which vertices are in R", and which are in R'. Thus the above checks are carried out easily. The code structure, and the corresponding complexities, look as follows:

For each Fragment	{ O(q) = O(p) }
Check Attachments	{ O(p) }
if necessary to update	
then Adjust Regions	{ O(q) = O(p) }

If we examine the above operations, we see that we may drop the complexity of the procedure Determine_Common_Regions down to $O(p^3)$. Checking the old fragments for updating requires scanning the attachments for each fragment $\mathcal{F}$ to see if any are in the old region or the new region (as remarked above, it is easy to do). Then we scan O(i+1) = O(q) = O(p) elements in the list of common regions to update them appropriately. There are O(p) attachments for each fragment, and we have O(q) = O(p) fragments. Thus we get a complexity of $O(p^2)$. This step of adjusting fragment values is done a total of O(p) times, and thus we get $O(p^3)$ steps.

Although this implementation is more efficient, there appears to be no direct method leading to a further reduction in complexity. The main problem with Algorithm 2.2 is that it does not use information inherent in the graph when it embeds the graph in the plane (no attempt is made at planning). In the next section, we discuss how Hopcroft and Tarjan develop a preprocessing of the graph, to generate enough information to embed each fragment as it is encountered, and further design a flexible data structure to cater for any changes in embeddings.

Section 2.2

The Hopcroft and Tarjan Fragment Addition Algorithm

The ideas on which the planarity testing algorithm we describe in this section are based come from an algorithm first proposed, in 1961, by Auslander and Parter [AP61], but their formulation was incorrect. In 1963 Goldstein [Gol63] published a corrected, efficient version of Auslander and Parter's planarity testing algorithm. In 1974 Hopcroft and Tarjan [HT74] derived a linear iterative technique using the ideas proposed by the previous authors. This last technique is the approach we shall consider. Firstly note that the algorithm is known as a fragment addition algorithm. By this we mean that an entire fragment is added to the subgraph H which we have built up so far. Even [Eve79] refers to the algorithm as a path addition algorithm and, although this is not incorrect (since we embed paths of the relevant fragment at a time), we use a different characterisation to distinguish from the "purer" path addition algorithm by Demoucron, Malgrange and Pertuiset.

In contrast to the Demoucron, Malgrange and Pertuiset Algorithm of the previous section we avoid any repetitive processing of edges. This is done as an initialisation step by generating a carefully chosen weighting function which we first apply to the edges of the graph. The edges incident with each vertex are then sorted so that they appear in non-decreasing order of their weights in the adjacency lists of the vertices. As we shall show, the information implicitly given by the ordering of the edges is sufficient to avoid any re-scanning of edges. Another feature of the algorithm is that we do not associate an embedding with the subgraph H completed at any stage of the algorithm. We reserve the freedom to efficiently move fragments from one region to another after they have been processed. Thus this algorithm differs markedly from the Demoucron, Malgrange and Pertuiset Algorithm.

The algorithm consists of two procedures, each of which is a Depth-First Search (DFS) of the graph. The first DFS orders the edges and is a modification of the DFS given in Chapter 1.

More specifically, on the set of vertices we compute a second lowpoint function in addition to the first lowpoint function. The second lowpoint is used to further order the edges incident from a vertex. For a vertex v, $L_1(v)$ and $L_2(v)$ denote the first and second lowpoint values respectively. From now on we use T to denote the DFS tree T of G. Let S(v) be the set of vertices u reachable from v by a (non-trivial) v-u path which uses tree edges followed by at most one back edge. Then, recall that for a vertex v of G,

$$L_1(v) = \min \{ \{dfi(v)\} \cup \{dfi(u) \mid u \in S(v)\} \}$$

If w is the unique vertex of T for which $dfi(w) = L_1(v)$, then the second lowpoint $L_2(v)$ is now defined as follows,

$$L_2(v) = \min \{ \{dfi(v)\} \cup \{dfi(u) \mid u \in [S(v) - \{w\}] \} \}$$

Intuitively $L_2(v)$ is the second smallest depth-first index value which can be reached from v by a path which uses tree edges followed at most one back edge. We can easily modify the DFS procedure to compute $L_2(v)$ values as well. Each time we backtrack to the parent or scan a new edge which is a back edge, as well as checking whether the first lowpoint $L_1(v)$ must be updated, we check whether $L_2(v)$ must be updated.

Suppose we are scanning a back edge $e = v \rightarrow u$, then

if $L_1(v) > dfi(u)$ then we assign $L_2(v) = L_1(v)$ and $L_1(v) = dfi(u)$.
if $L_1(v) < dfi(u)$, then we assign $L_2(v) = \min \{L_2(v), dfi(u)\}$.

If we are backtracking to the parent of v, namely Parent(v), then we must update the lowpoint values of Parent(v) according to the values of v.

If $L_1(v) < L_1(Parent(v))$, then we assign

$L_2(Parent(v)) = \min \{L_2(v), L_1(Parent(v))\}$ and
$L_1(Parent(v)) = L_1(v)$.

If $L_1(v) = L_1(Parent(v))$, then we set
$L_2(Parent(v)) = \min \{L_2(v), L_2(Parent(v))\}$.

Lastly if $L_1(v) > L_1(Parent(v))$, then
$L_2(Parent(v)) = \min \{L_1(v), L_2(Parent(v))\}$.

These modifications are simple to implement, and consist of the insertion of a few "if statements" into the DFS procedure's code. The complexity of the DFS is not changed, and is thus still linear. Consider Figure 2.5, below, that shows a DFS of a graph.

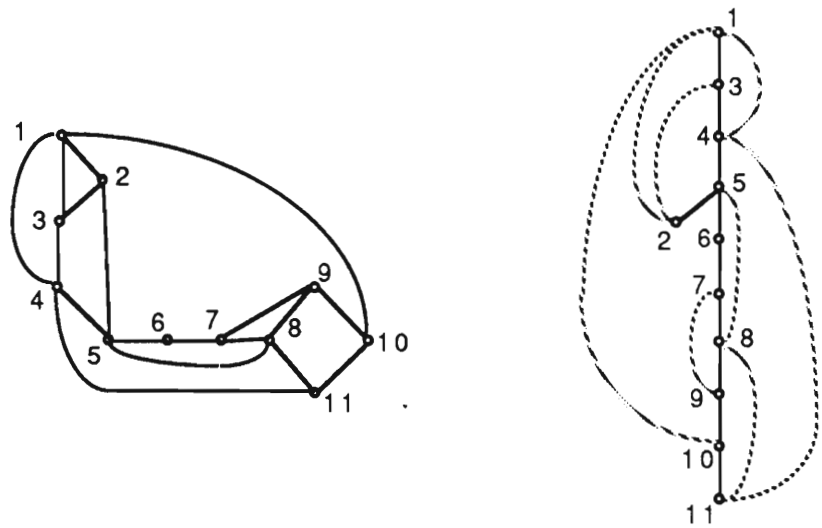


Figure 2.5 : A graph G and its associated DFS tree
(back edges are drawn as dashed lines)

From Figure 2.5 we may determine, for each vertex v, the corresponding lowpoints $L_1(v)$ and $L_2(v)$. Table 2.1, below, lists these values.

Vertex v	dfi(v)	$L_1(v)$	$L_2(v)$
1	1	1	1
2	11	1	2
3	2	1	2
4	3	1	2
5	4	1	2
6	5	1	3
7	6	1	3
8	7	1	3
9	8	1	3
10	9	1	3
11	10	3	7

Table 2.1 - Vertices, DFS indices and Lowpoints

Once the DFS computes the two lowpoints of a vertex, it is possible to sort the edges. For every edge $v \xrightarrow{e} u$ we generate a weighting $\phi(e)$ as follows :

$$\phi(e) = \begin{cases} 2.dfi(u) & \text{if } v \xrightarrow{e} u \text{ is a back edge} \\ 2.L_1(u) & \text{if } v \xrightarrow{e} u \text{ is a tree edge and } L_2(u) \geq dfi(v) \\ 2.L_1(u) + 1 & \text{if } v \xrightarrow{e} u \text{ is a tree edge and } L_2(u) < dfi(v) \end{cases}$$

Now we order the edges in the adjacency lists of each vertex in non-decreasing order of $\phi(e)$. By use of a simple distribution sort (see for example Knuth [Knu73]), we may perform this in linear time. We arrange a series of $2p + 1$ buckets numbered 1, 2, .., $(2p + 1)$. We remove each edge e from the adjacency lists of the vertices and compute $\phi(e)$. The edge is then placed into bucket $\phi(e)$. When the adjacency list of each vertex is empty, we start from bucket $2p + 1$ and for each bucket we add each edge in that bucket to the head of the adjacency list of the vertex which contained this edge in its adjacency list previously. By inserting

the edges at the head of the adjacency list, the edges in the adjacency list will be sorted in non-decreasing order of $\phi(e)$. The overall complexity of this step is thus O(q). Now we are ready to present the second procedure of the algorithm, a modified DFS. We follow the approaches of Even [Eve79], Mehlhorn [Meh84] and Hopcroft and Tarjan [HT74].

The first thing we do in the second part of the algorithm is to find a cycle C which contains the root vertex v_1 ($dfi(v_1) = 1$). The cycle C may be determined by starting at the root v_1 and selecting the first edge on the adjacency list of each vertex, until we encounter a back edge. The weighting function $L_1(v)$ and the fact that G is 2-connected guarantee that we will eventually form a path of tree edges followed by a single back edge to the root v_1 (see Chapter 1 for more details). Consider any fragment $\mathcal{F}$. There are only two possible positions for this fragment in the embedding of G, i.e. inside the cycle or outside. The algorithm efficiently determines if it is possible for the fragments to be placed around the cycle in such a way that the resulting embedding $\hat{G}$ is planar.

After we have found the cycle the remainder of the second procedure may be further separated into two distinct parts for every fragment $\mathcal{F}$. Suppose we are trying to add $\mathcal{F}$ to the subgraph H built so far. The first part recursively tests if the subgraph induced by the edges of C and $\mathcal{F}$ is planar. Then, the second part of the algorithm "merges" $\mathcal{F}$ with the rest of the fragments found so far. The merging process examines the attachments of the other fragments and arranges fragments on the inside and outside of C, such that no fragments on the same side of the cycle interlace.

For example, in the graph G as shown in Figure 2.5 we have that our original cycle C may be C : $v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_1$. The fragments of G with respect to C are shown in Figure 2.6, below. Note that $\mathcal{F}_2$ and $\mathcal{F}_3$ interlace, and consequently may not both be placed on the inside or both on the outside of C. Also $\mathcal{F}_3$ and $\mathcal{F}_5$ interlace, and thus the placement of one of the fragments $\mathcal{F}_2$, $\mathcal{F}_3$ or $\mathcal{F}_5$ automatically determines the placement of the other two fragments.

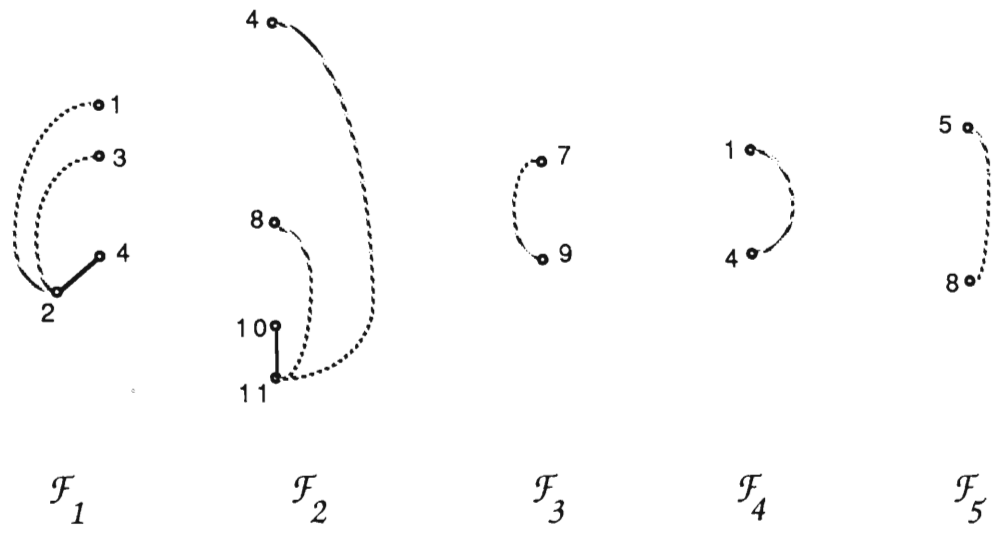


Figure 2.6 : The Fragments of G in Figure 2.5 with cycle C :v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_1

We shall now proceed to describe the second part of the second procedure. The main loop of the algorithm is a path generation procedure which generates an initial path of every fragment $\mathcal{F}$. This first path has very important properties, as we shall show later. To generate a path P from some vertex v_k on C we start with an edge which is not in C and not already explored. At each stage we add to P the first edge in the adjacency list of the current vertex which has not already been explored. Thus we merely follow unexplored tree edges down the tree until the first unexplored edge in the adjacency list of the current vertex is a back edge. This unexplored back edge reaches a vertex v_j say in C which is the end vertex of P. Algorithm 2.3 below details the algorithm so far. Note that the cycle finding part of the algorithm is a special case of the path generation.

Algorithm 2.3: Cycle_and_Path_Generation

{ Generate a cycle and the initial paths of every fragment $\mathcal{F}$ }

```
Mark all edges unused
PC = v1                                  { path only has the root to start }
v = v1                                   { current vertex is the root }
Repeat                                   { generate initial cycle }
    Let the first edge incident with v be e = v → vi
    Mark e as used { we add it to the initial path }
```

```
        if v_i ≠ v_1
            then
                P_C = P_C + v_i          { add to the path }
                v = v_i                  { and current vertex is adjusted }
    until (v_i = v_1)
    Output C : P_C followed by vv1       { use P_C and edge vv1 to form cycle }

    while v ≠ v_1 do                     { generate paths }
        if no edges incident from v are unused
            then v = Parent(v)           { backtrack up C }
            else                         { start a new path }
                P = v
                Let the first unused edge incident with v be e = v → v_i
                while e = v → v_i is not a back edge do
                                         { build path of tree edges }
                        Mark e used
                        P = P + e        { add to the path }
                        v = v_i          { adjust current vertex }
                P = P + e                { add the back edge to the path }
                Output P
```

end

A study of the above algorithm will reveal that the algorithm is essentially a DFS on the graph and that all properties of a DFS of a graph described in Chapter 1 will apply in the following discussion.

Consider, as an example of Algorithm 2.3, the cycle and paths generated if we give, as input to Algorithm 2.3, the graph G shown in Figure 2.5. The vertices of the cycle generated are v_1, v_3, v_4, v_1. The vertices of the paths which were generated by Algorithm 2.3 are

$P_1 : v_4, v_5, v_2, v_1$
$P_2 : v_2, v_3$
$P_3 : v_5, v_6, v_7, v_8, v_9, v_{10}, v_1$
$P_4 : v_{10}, v_{11}, v_4$
$P_5 : v_{11}, v_8$
$P_6 : v_9, v_7$
$P_7 : v_8, v_5$

Lemma 2.2: Algorithm 2.3 correctly finds a cycle C in G

Proof: Since G is 2-connected, and from the reordering of the adjacency lists, for any vertex v, the edge e = v → u such that $L_1(u) = 1$ will be chosen first. Thus, from Chapter 1 and from the discussion about cycle generation in the preceding section, we know that we will eventually complete the required cycle C. ❑

The important part of the algorithm is the path generation. There are several characteristics of the walks generated which are crucial to the successful operation of the Hopcroft and Tarjan Algorithm. Next we will prove and discuss the properties of the walks P generated by Algorithm 2.3.

Lemma 2.3: Each walk P generated by Algorithm 2.3 is indeed a path, and has only two vertices in common with previously generated paths, namely the first v_f and the last v_ℓ.

Proof: That the walk P is a path follows directly from the fact that we are dealing with a DFS tree. We start with some old vertex v_f (a new vertex is one that is not incident with used edges) and proceed along tree edges e = v → u (necessarily dfi(v) < dfi(u)) until we reach a back edge. Because G is 2-connected, all paths do end in back edges. Furthermore, we also know that, for the same reason, if e = $v_f \rightarrow v_\ell$ is the back edge, then dfi(v_ℓ) < dfi(v_f), and so v_ℓ is a vertex different from the other path vertices. We have used DFS tree edges and only one back edge to construct P, thus P is a path.

Recall from Chapter 1, since Algorithm 2.3 is a DFS, all ancestors of an old vertex are old, and so v_ℓ is old. Because each edge which we choose is unused and we are performing a DFS, that part of the tree rooted at v_f, and having the first edge chosen as unexplored is itself unexplored. Thus the internal vertices before the back edge must be new. ❑

Define S_v to consist of the descendants of v, together with v. Then, the following lemma will prove useful in subsequent discussions.

Lemma 2.4: Let v_f and v_ℓ be the first and last vertices of a path P generated by Algorithm 2.3, and let $v_f \xrightarrow{e} v$ be the first edge of P. Then,

(i) if $v \neq v_\ell$ then $L_1(v) = dfi(v_\ell)$, and

(ii) v_ℓ is the vertex with the lowest DFS index reachable from S_{v_f} via a back edge which has not been marked used.

Proof: Let $S_{v_f} = A \cup B$, where B represents those vertices from which we have already backtracked prior to the start of the generation of the current path, and $A = S_{v_f} - B$. Since we only backtrack from a vertex once all incident edges are used, we have $v_f \in A$. Let u be the vertex with lowest DFS index reachable, from a vertex in B, via an unused back edge. Because the adjacency lists are sorted, the first unused edge in the adjacency list of v_f is the initial edge on a directed path to u which uses either a single back edge to u or a series of tree edges followed by a back edge. Thus $v_\ell = u$ and the lemma follows. ❑

The next lemma is an immediate consequence of the preceding result.

Lemma 2.5: Suppose P_1 and P_2 are two paths whose first and last vertices are f_1, l_1 and f_2, l_2 respectively. If P_1 is generated before P_2 and f_1 is an ancestor of f_2, then $dfi(l_1) \leq dfi(l_2)$.

The significance of this result is its usefulness in obtaining the following fact: If the set of attachments of a fragment $\mathcal{F}$ on the cycle C produced by Algorithm 2.3, are $w_0, ..., w_r$; with $dfi(w_0) < dfi(w_1) < ... < dfi(w_r)$, then, as we shall prove later, all the edges of $\mathcal{F}$ incident with an attachment w_i ($1 \leq i \leq r$) are back edges except for an edge incident with w_r. Lemmas 2.4 and 2.5 imply that the path generation algorithm will produce a path $P = w_r, w_{r+1}, ..., w_k, w_0$, which has only the vertices w_r and w_0 in common with C. See Figure 2.7. The path has first and last vertices the attachments of $\mathcal{F}$ with largest and smallest DFS index respectively. Mehlhorn [Meh84] defines the spine cycle $SC(\mathcal{F})$ of $\mathcal{F}$ with respect to C and P as follows. Let $w_0, ..., w_k$ be as above, then $SC(\mathcal{F})$ consists of the w_0-w_r path on C, together with the path P.

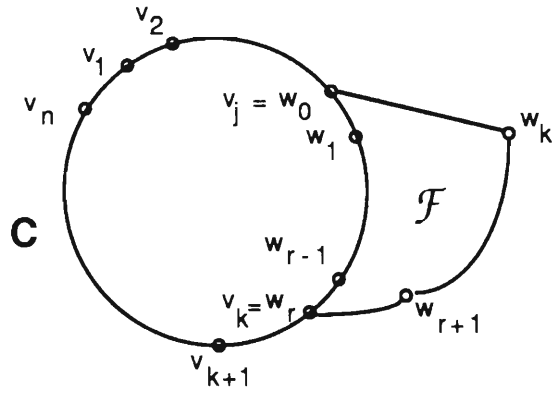


Figure 2.7 A cycle C with fragment F

The importance of the spine cycle is that it clearly defines the procedure we follow. The recursive application of the algorithm to a fragment $\mathcal{F}$ considers a new cycle, namely SC($\mathcal{F}$), and the set of fragments F of $\mathcal{F}$ with respect to SC($\mathcal{F}$). The rest of the cycle C in the original graph C + $\mathcal{F}$ may be thought of as an external fragment with respect to SC($\mathcal{F}$). In Figure 2.7, the external fragment is the path P' : $w_r = v_k, v_{k+1}, ..., v_n, v_1, v_2, ..., v_j = w_0$. We note that every fragment $\mathcal{F}_i \in F$ necessarily has attachments on the initial path P = $w_r, w_{r+1}, ..., w_k, w_0$. Further, SC($\mathcal{F}$) has the property that every fragment of $\mathcal{F}_i \in F$ with attachments on $w_0, w_1, ..., w_r$ must necessarily be placed in the inside of SC($\mathcal{F}$), because $\mathcal{F}_i$ interlaces with our external fragment P'.

Let us now consider what happens when two paths have the same starting and ending vertices. We use the second lowpoint $L_2(v)$ to characterise the paths that are generated.

Lemma 2.6: Let P_1: $v_f \xrightarrow{e_1} u_1 ... \rightarrow v_\ell$ and P_2: $v_f \xrightarrow{e_2} u_2 ... \rightarrow v_\ell$ be two generated paths, where P_1 is generated before P_2.
If $u_1 \neq v_\ell$ and $L_2(u_1) < dfi(v_f)$ then $u_2 \neq v_\ell$ and $L_2(u_2) < dfi(v_f)$.
Proof: By the definition of ø(e), $ø(e_1) = 2\,L_1(v_\rho) + 1$. Since e_2 appeared after e_1 in the adjacency list of f, we must have that $ø(e_1) \leq ø(e_2)$. If $u_2 = v_\ell$, then $v_f \xrightarrow{e_2} u_2$ is a back edge. Thus, $ø(e_2) = 2.dfi(u_2) = 2.dfi(v_\rho) < 2.dfi(v_\rho) + 1 = ø(e_1)$, which is impossible. Now, if $L_2(u_2) \geq dfi(v_f)$, then $ø(e_2) = 2.dfi(v_\rho) < 2.dfi(v_\rho) + 1 = ø(e_1)$, which is also impossible. ❑

The reason for this lemma will become apparent later. Now, consider the cycle C: $v_1, v_2, ..., v_n, v_1$. Since we are performing a DFS, $dfi(v_1) < dfi(v_2) < ... < dfi(v_n)$. Now, let us consider the structure of a fragment $\mathcal{F}$ with respect to C.

Lemma 2.7: Let $\mathcal{F}$ be a fragment of G with respect to C which is not a single back edge. As in Figure 2.7, let the attachments of $\mathcal{F}$ be $w_0, w_1, ..., w_r$, and let the path produced by the path generation algorithm be $P = w_r, w_{r+1}, ..., w_k, w_0$. Then all edges of $\mathcal{F}$ incident with $w_0, w_1, ..., w_r$ are back edges except for the edge $e = w_r \rightarrow w_{r+1}$.

Proof: The proof follows directly from the fact that we are performing a DFS. We find the original cycle C, then we only backtrack from a vertex v once all edges incident with v have been marked used. No internal part of $\mathcal{F}$ could have been scanned when we backtrack along the cycle C into w_r. There must be a tree edge going into $\mathcal{F}$. If there was not such a tree edge, then all edges incident with $\mathcal{F}$ are back edges which is impossible. Since G is 2-connected, all paths generated have their end vertices higher in the tree (i.e. with lower DFS index) than the start vertices. Thus $w_r w_{r+1}$ is a tree edge. Furthermore, it is the only tree edge on C incident with $\mathcal{F}$. To see this note that if there was another tree edge $w_i \rightarrow u$ from C incident with a vertex w_i of $\mathcal{F}$, then there is a path from w_r to u in $\mathcal{F}$. This path would have been explored before backtracking up C from w_r. Now, we only backtrack from u once all incident edges have been marked used and we only explore edges incident from w_i, not on C, once we have backtracked from w_r along the cycle to w_i. Thus, the edge $e = w_i u$ would have been explored from u, and directed to w_i as a back edge. ❑

Corollary 2.1: Once a fragment $\mathcal{F}$ is entered it is completely traced out before we backtrack out of $\mathcal{F}$.

The proof of this fact follows directly from Lemma 2.7.

We can now turn our attention to the problem of deciding where we can place a fragment. We define the root of a fragment $\mathcal{F}$ to be the unique vertex w_r on the cycle C which is incident with a tree edge which belongs to $\mathcal{F}$.

Because of our path generation technique we have the convenient situation where all fragments starting from vertices with DFS index greater than or equal to that of the root of the current fragment $\mathcal{F}$, have already been explored in our DFS search. Thus we know the attachments of each of these fragments, and the restrictions on their embeddings due to the way they interlace. Furthermore, we know the attachments of the current fragment $\mathcal{F}$ with greatest and smallest DFS index. The following result provides a clear answer as to whether we may place, at this stage, the first path of the fragment, and hence the entire fragment, inside or outside the cycle.

Lemma 2.8: Let P: $v_k = u_1, u_2, \ldots, u_m = v_j$ be the first path of $\mathcal{F}$ generated by the algorithm (where of course $dfi(v_j) < dfi(v_k)$). Then, P may be placed on the inside (outside) of C if and only if there is no back edge $v \rightarrow w$ ($dfi(v) > dfi(w)$) already drawn inside (outside) the cycle for which $dfi(v_j) < dfi(w) < dfi(v_k)$.

Proof: Without loss of generality we assume that we are to embed $\mathcal{F}$ inside C. Suppose no such back edge exists. Then we may freely embed F inside C (although, of course, at a later stage of the algorithm this choice could be prohibited). To see this, note that no tree edge lying between v_j and v_k could have been explored yet since we only backtrack up C from v_k once all edges incident from v_k have been explored.

For the converse, suppose there is a back edge $v \xrightarrow{e} w$ with $dfi(v_j) < dfi(w) < dfi(v_k)$. Then there are two cases. Let $\mathcal{F}'$ be the fragment to which this back edge e belongs. The two cases depend on the position of the root w_f of $\mathcal{F}'$.

Case 1: Suppose $dfi(w_f) > dfi(v_k)$; see Figure 2.8a. Then we have a sequence of four vertices w_f, v_k, w, v_j appearing around C in that order. Thus, P and $\mathcal{F}'$ interlace and hence cannot be drawn on the same side of C.

Case 2: Suppose $dfi(w_f) \leq dfi(v_k)$. Then, $dfi(w_f) = dfi(v_k)$, since $dfi(v_k) \leq dfi(w_f)$. Hence, $w_f = v_k$. Let the first path of $\mathcal{F}'$ to be generated be $P' = w_f, w_{f+1}, \ldots, w_m, w_0$. We split this option into a further two cases.

Subcase 2.1: Suppose first that $dfi(w_0) \neq dfi(v_j)$. Since P' was generated before P, it follows, from the edge ordering, that $dfi(w_0) \leq dfi(v_j)$. So $dfi(w_0) < dfi(v_j)$. Now, since $\mathcal{F}'$ has another fragment w between v_k and v_j on C, $\mathcal{F}'$ and $\mathcal{F}$ must interlace (see Figure 2.8b). So, in this case, P cannot be placed on the same side of C as the back edge $v \xrightarrow{e} w$.

Subcase 2.2: Suppose $dfi(w_0) = dfi(v_j)$. Then, $w_0 = v_j$. Note that P' cannot be a single back edge, otherwise, $\mathcal{F}' = P'$, $w = w_f$, and thus $dfi(w) = dfi(v_k)$, contrary to our hypothesis. Since $w_f w_{f+1}$ is a tree edge, and since there is a w_f - w path in $\mathcal{F}'$ which uses only tree edges followed by one back edge, there is a w_{f+1} - w path in $\mathcal{F}'$ which uses only tree edges followed by one back edge. Since $L_1(w_{f+1}) \leq dfi(w_0) < dfi(w)$, it now follows that $L_2(w_{f+1}) \leq dfi(w)$, so $L_2(w_{f+1}) < dfi(v_k)$. Thus, by Lemma 2.6, P is not a single back edge and $L_2(u_2) < dfi(v_k)$. Hence, $\mathcal{F}$ must have a third attachment v_z say. Thus, either $\mathcal{F}$ and $\mathcal{F}'$ share three attachments ($v_z = w$, see Figure 2.8c), or they have four attachments which appear in order around C, such that $dfi(v_j) < dfi(w) < dfi(v_z) < dfi(v_k)$ (see Figure 2.8d). So, once again, $\mathcal{F}$ cannot be embedded on the same side of C as $\mathcal{F}'$. Hence, $\mathcal{F}$ cannot be placed on the same side of C as $v \xrightarrow{e} w$. Consequently P cannot be placed on the same side of C as $v \xrightarrow{e} w$. ❑

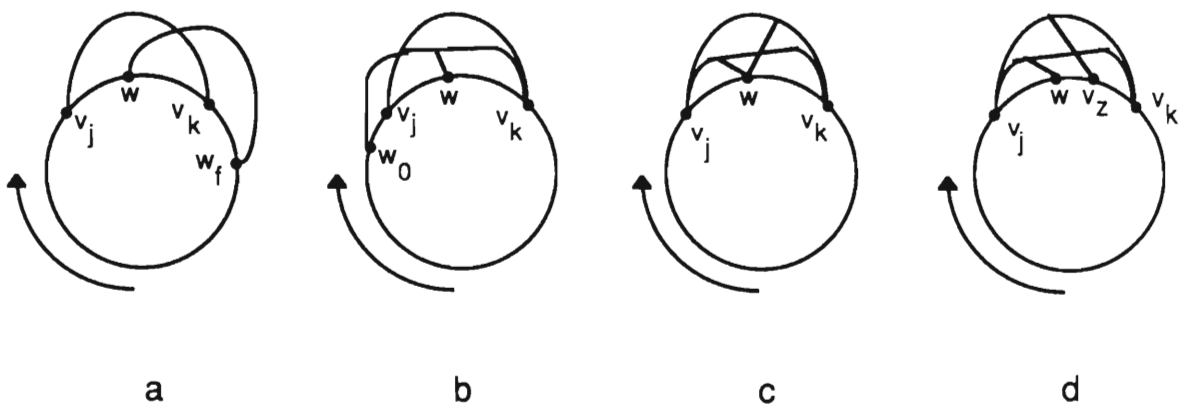


Figure 2.8 : The four different cases in Lemma 2.8

Now we may use the previous results to build up a more detailed algorithm to test planarity. We use the results of Chapter 1 to test for planarity. We know from these results that G is planar if and only if:

> (i) for every fragment $\mathcal{F}$ of G with respect to C, C + $\mathcal{F}$ is planar, and
> (ii) the set of fragments of G with respect to C can be partitioned into two sets such that no two fragments in the same set interlace.

By Lemma 2.8 we can test whether the part of the graph explored so far is planar, since Lemma 2.8 provides us with a condition to test if we may partition the set of fragments drawn so far into two sets such that the new fragment $\mathcal{F}$ does not interlace with any other fragment in the same set. We may then use a recursive generation of the algorithm to test if C + $\mathcal{F}$ is planar. To aid us in our discussions on the validity of condition (ii), above, we observe the following. Let H be the bipartite graph whose vertices correspond to fragments of G with respect to C, such that two vertices are adjacent if the corresponding fragments interlace. Then, condition (ii) is satisfied if and only if H is bipartite.

Suppose as usual that the set of attachments of a fragment $\mathcal{F}$ are w_0, ..., w_r; with $dfi(w_0) < dfi(w_1) < ... < dfi(w_r)$, and $SC(\mathcal{F})$ is the w_0-w_r path on C together with the path w_r, w_{r+1}, ..., w_0. As discussed earlier, if we can successfully arrange the fragments so that all fragments of $\mathcal{F}$ with respect to $SC(\mathcal{F})$ with attachments in the set $\{w_1, ..., w_{r-1}\}$ are placed on the inside of the spine cycle $SC(\mathcal{F})$, then C + $\mathcal{F}$ is planar. If the recursive step to test the planarity of C + $\mathcal{F}$ fails, or if we cannot arrange the fragments as described above in (ii), then G is non-planar. The complete algorithm is shown below in Algorithm 2.4. We generate the cycle first, and enter the algorithm with Current being the last vertex in the initial cycle generated in Algorithm 2.3, and Root the root of the DFS tree.

Algorithm 2.4: Recursive_Test_Planarity (Root, Current)

```
{ Recursively test Planarity of a graph G }
(Root is root of the part of the fragment we are exploring,
        and Current is the current vertex we are at )

while Current ≠ Root do

{ that is, while we have not backtracked to root of DFS tree }

    while Current is incident with unused tree edges do

{ Generate first path of frag. which starts at Current and ends at Next, }
{ where Next is the last vertex before the back edge }

        Generate_Path (P, Current, Next)

{ Recursively call Algorithm 2.4 to test planarity of fragment and cycle }
{ with the new Root = Current, and the new Current = Next }

        Recursive_Test_Planarity ( Current, Next)

        if not Planar
            then Halt       { end the entire program }
            else
                Arrange_Fragments_After_Recursive_Call
                        { Arrange so that no two fragments interlace }
                if not Planar
                    then Halt

{ at this stage all edges incident with Current are scanned }
{ so we backtrack along cycle to parent }

    Current = Parent(Current)
end { of while }
```

end

Firstly, note that for every tree edge incident with Current_Vertex, we start a new fragment, $\mathcal{F}$ say. Procedure Generate_Path generates a path using the latter part of Algorithm 2.3 and returns the last new vertex of the path generated (i.e. the vertex which the back edge of the generated path is incident from; this is the last vertex in the spine cycle SC($\mathcal{F}$), i.e. namely, vertex w_k, as shown in Figure 2.7). We then need to generate the fragments of the current fragment $\mathcal{F}$. We do this by a recursive call to Algorithm 2.4, this time the Root is Current, the vertex which we were working with, and the new Current is w_k. If the call was unsuccessful,

then the graph is not planar. Otherwise, in procedure Arrange_Fragments_After_Recursive_Call we try to arrange the fragments SC($\mathcal{F}$) so that all fragments with attachments on the path w_1, ..., w_{r-1} of SC($\mathcal{F}$) are placed on the inside of SC($\mathcal{F}$). If this is not possible, then we fail the planarity test, since they interlace with our external fragment, namely the rest of the cycle C. Notice also that we only backtrack from a vertex when we have exhausted all tree edges.

The key to the efficiency of the algorithm lies, as in most linear algorithms, in our choice of data structures. Hopcroft and Tarjan introduced three stacks to maintain the fragments efficiently. We call the stacks Inner, Outer and Blocks. Inner and Outer are two doubly linked lists which we use to store in ascending (non-decreasing) sorted order the attachments of the fragments embedded so far. As the names suggest, Inner and Outer store the attachments for the fragments stored on the inside and outside of the cycle C respectively. Each element in the stack Blocks contains two pointers, which point to vertices on Inner and Outer respectively (we shall discuss Blocks more fully later).

When we backtrack into a vertex v_k for the first time, we may remove all occurrences of v_k from the stacks Inner and Outer, and all pointers on Blocks to v_k on Inner and Outer. As can be seen from Lemma 2.8, the decision on where to place the current fragment $\mathcal{F}$ depends entirely on the back edges strictly in between the first and last attachments of $\mathcal{F}$ on C. Every attachment v on Inner and Outer with $dfi(v) \geq dfi(v_k)$ is therefore redundant. Thus, the only entries on the two stacks Inner and Outer are attachments incident with back edges. Also, more importantly, note that when we backtrack from the recursive step, the only fragment attachments of the recursion call left on the stacks are those vertices w with $dfi(w_0) < dfi(w) < dfi(w_r)$. Thus the step of checking that all fragments of $\mathcal{F}$ with respect to SC($\mathcal{F}$) which have attachments on the cycle C may be placed inside SC($\mathcal{F}$), reduces to checking if all fragments of $\mathcal{F}$ with respect to SC($\mathcal{F}$), left on the stacks may be placed inside SC($\mathcal{F}$).

Suppose that, at any stage in the algorithm, we have determined fragments $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_k$, so that we may partition these fragments into

two sets so that no two fragments in the same set interlace. Then the graph H_k, whose vertex set is $\{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k\}$, and which is such that two vertices are adjacent if and only if the corresponding fragments interlace, is bipartite. After the k-th fragment of G with respect to C has been determined, a *fragment block* (at this stage) consists of the attachments of the vertices of a component of H_k which appear on Inner and Outer. The stack Blocks keeps track of the fragment blocks created so far. As we shall see below, by maintaining the list of fragment blocks we can avoid processing of single fragments.

Lemma 2.9: Let F be a fragment block whose element with highest DFS index is v_h and whose element with lowest DFS index is v_ℓ. Then, if v_f is a vertex on either Inner or Outer and if $dfi(v_\ell) < dfi(v_f) < dfi(v_h)$, then v_f belongs to F.

Proof: The proof proceeds by induction on the number of times we have updated fragment blocks. If $\mathcal{F}$ is the first fragment to be tested, then the lemma holds since we have one fragment block whose elements are precisely attachments of $\mathcal{F}$ incident with all the back edges of $\mathcal{F}$. Suppose the lemma holds after we have updated fragment blocks for the k-th time. Let the attachments of $\mathcal{F}$ with lowest and highest DFS index be v_j and v_k, respectively. Then we have two cases.

Case 1: There is no vertex v_f on Inner or Outer stacks for which $dfi(v_j) < dfi(v_f) < dfi(v_k)$. Then there are no back edges drawn between any attachments of $\mathcal{F}$. Thus, by Lemma 2.8, we may place $\mathcal{F}$ inside or outside C, i.e. $\mathcal{F}$'s attachments on Inner or Outer form a fragment block and the lemma still holds.

Case 2: Suppose that there are vertices of Inner and Outer with DFS index strictly in between the DFS indices of v_j and v_k. Then, by Lemma 2.8, the fragments they are attachments of interlace with $\mathcal{F}$. Thus, they belong to the same fragment block, since these fragments will now be merged in H_{k+1}. We must thus merge all the old fragment blocks to which these fragments belong into one new fragment block. Let v_ℓ and v_h be the vertices with lowest and highest DFS index in the new block. Now, suppose that the old fragment block which had v_ℓ as an attachment, had v'_h as its vertex with greatest DFS index. Certainly $dfi(v_j) < dfi(v'_h)$ (or else

fragments in this fragment block would not have interlaced with $\mathcal{F}$). Now, by the inductive hypothesis, we know that all attachments with DFS indices between the DFS indices of v'_h and v_ℓ, which are on Inner and Outer, belong to the same fragment block, and so, after the merging operation all attachments on Inner and Outer, with DFS index between v'_h and v_ℓ, belong to the new fragment block.

Similarly, suppose that the old fragment block which had v_h as an attachment had v'_ℓ as its vertex with lowest DFS index. Again, $dfi(v'_\ell) < dfi(v_k)$, and all attachments with DFS index between the DFS indices of v_ℓ and v_h belong to the new fragment block.

Lastly, consider any of the old fragment blocks with v_o and v_m ($dfi(v'_h) \geq dfi(v_m) \geq dfi(v_o) \geq dfi(v'_\ell)$) as its vertices with lowest and greatest DFS index. It is easy to see that all vertices with DFS index between v_m and v_o belong to the new fragment block, because the fragments they are attachments of interlace with $\mathcal{F}$. ❑

From Lemma 2.9 we now have a procedure for the maintenance of fragment blocks. When we have the first path $P : v_k \rightarrow \ldots \rightarrow v_j$ of our fragment $\mathcal{F}$, to decide where to place the path, we look at the top elements t_i and t_o of the Inner and Outer stacks respectively. As a convention, we shall always place the new fragment on Inner. There are four possible cases:

(i) If $dfi(v_j) \geq dfi(t_o)$ and $dfi(v_j) \geq dfi(t_i)$, then no merger between blocks is necessary. That this is true is easy to see if we consider that t_o and t_i are the highest numbered vertices of the fragment blocks created so far. If the lowest numbered vertex of $\mathcal{F}$ is at least as large as any attachment of a fragment found so far, then we know that $\mathcal{F}$ may be placed either inside or outside C, since no back edges fit the description of Lemma 2.8. We enter the attachments of $\mathcal{F}$ as a new block on Inner.

(ii) If $dfi(v_j) < dfi(t_o)$ but $dfi(v_j) \geq dfi(t_i)$, then we may still place P in the inside of C. We first need to merge all fragment blocks which have attachments w with $dfi(v_j) < dfi(w) < dfi(t_o)$. Intuitively this is because we are now forcing those fragment blocks on Outer with attachments w such that $dfi(v_j) < dfi(w) < dfi(t_o)$ to be embedded on

the other side of the cycle that $\mathcal{F}$ is embedded on. There is no need to check Inner, but still we may merge several blocks which have attachments on Outer. We then place the attachments of $\mathcal{F}$ on Inner in non-decreasing order.

(iii) If $dfi(v_j) < dfi(t_i)$ but $dfi(v_j) \geq dfi(t_o)$, then we perform the following operations. We first need to merge all fragment blocks which have attachments w with $dfi(v_j) < dfi(w) < dfi(t_i)$. There is no need to check Outer, but still we may merge several blocks which fit this criteria. We *switch sections* of the new fragment block by placing all attachments of the fragment block which are on Inner onto Outer, and all attachments of the fragment block which are on Outer onto Inner. Thus the fragments of the new fragment block which were on the inside of C are now placed on the outside, and vice versa. Lastly, we place the attachments of $\mathcal{F}$ on Inner in non-decreasing order.

(iv) If $dfi(v_j) < dfi(t_i)$ and $dfi(v_j) < dfi(t_o)$, then the case is more complex. Here blocks on both Inner and Outer interlace with $\mathcal{F}$. As before, we need to merge the fragment blocks with attachments on Inner and Outer whose DFS indices are greater than $dfi(v_j)$. However, we cannot just merge the fragment blocks. We need to ensure that it is possible for $\mathcal{F}$ to be placed in the graph at all. We check each fragment block one at a time. If the highest entry on Inner for the fragment block is strictly greater than $dfi(v_j)$, then we switch sections for that fragment block. If the highest entry is still greater, then we halt and declare that the graph is non-planar, since fragments of that block interlace with F both on the inside and the outside of the cycle. Otherwise we continue to the next fragment block. If these switches succeed, then we have an arrangement where we can place our attachments of $\mathcal{F}$ on the Inner stack, again in non-decreasing order.

It is important to note that the switching of sections of the fragment blocks may be efficiently performed using the third stack Blocks. Every element in the stack Blocks, has two pointers, Inner_Ptr and Outer_Ptr which point to the lowest attachments for a particular fragment block on Inner and Outer respectively. Note that, in the discussion above, the new

fragment block we create is always on top of the stacks Inner and Outer, so that a pair of pointers on Blocks is sufficient. That is, we do not need another pair of pointers to the attachments of greatest DFS index of a fragment block. Thus, to switch sections of a fragment block, we can immediately access the lowest attachments of that fragment block, and a simple pointer manipulation will modify the lists and hence perform the switch. See Figure 2.9, below. Immediately after performing a single pointer switch, we have swopped all elements of a fragment block on stacks Inner and Outer. To merge two adjacent fragment blocks on Blocks, we merely discard the top element out of the two that we want to merge. A little care must be exercised throughout for the case where a block has an empty Inner or Outer section. In this case we set the pointer to nil. When we merge blocks in this case, we set Inner_Ptr or Outer_Ptr to the lowest non-empty pointer on the relevant section.

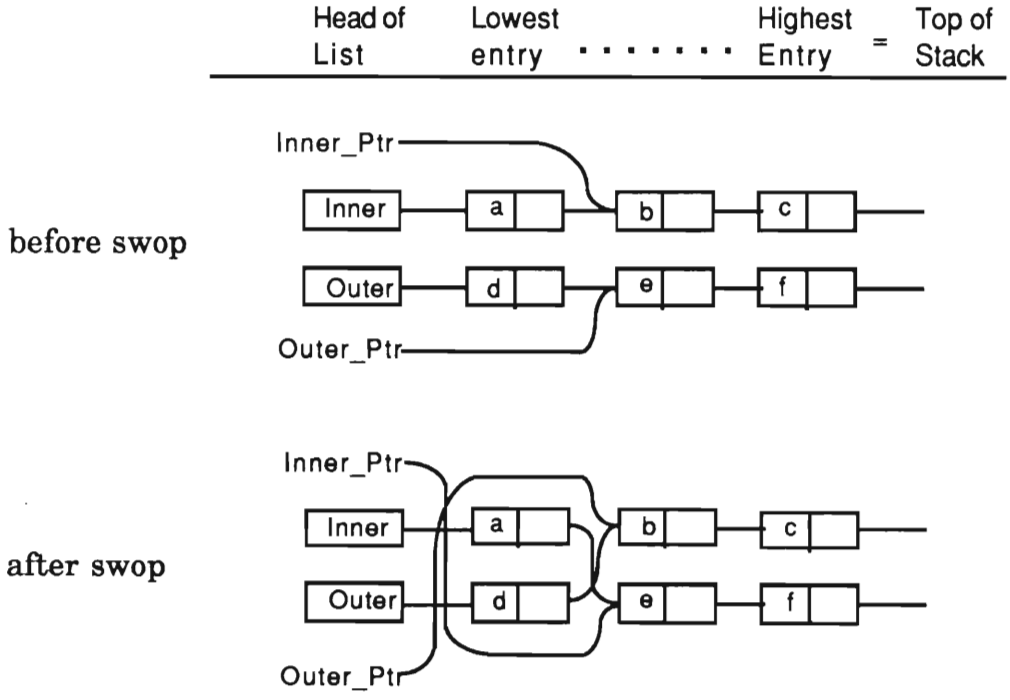


Figure 2.9 : Swopping entries on Inner and Outer via pointers on Blocks

To remove occurences of attachments from Inner and Outer, we merely remove the occurences from the top of the stack. To remove the occurrence from Blocks, we merely set the relevant pointer (i.e. the

pointer to either Inner or Outer) to nil. If the element then contains two empty pointers, then we may remove it entirely from Blocks.

The recursive step to test planarity of a fragment $\mathcal{F}$ and corresponding spine cycle SC($\mathcal{F}$) is handled easily via a special end-of-stack marker. We place this marker on Outer before we make the recursive call. Then, when we have finished the call, all the attachments of fragments that the recursive step generated are either on Inner or above the marker on Outer. From the earlier discussions we know that all the new fragments must be merged and placed on Inner. Thus we attempt to merge the fragments on Outer and Inner together onto Inner until we expose the end of stack marker on Outer. We remove the marker and merge all the fragment blocks which were created in the recursion. Then we merge this block with the initial block on Inner which contains the vertex v_k which was formed when the first path of $\mathcal{F}$ was generated. Then we continue our planarity testing. Algorithm 2.5 gives the algorithm. We enter Algorithm 2.5 with a cycle C, Root = First Vertex on C and Current = Last Vertex on C.

Algorithm 2.5: Test_Planarity (Root, Current)

{ Test planarity of a subgraph of a graph G, rooted at }
{ Root, and with current vertex to test from is Current }

```
while Current ≠ Root do
{ while we have not backtracked to root of DFS tree }

    while Current has unused tree edges do
{ cleanup the stacks }
        Delete entries v on Inner, Outer, Blocks with dfi(v) ≥ Current
{ Generate first path of fragment - starts at Current, ends at Next }
        Generate_Path (P, Current, Next)
{ Check for conflicting blocks, as discussed above }
        While Position of top entry on Blocks determines P
            Delete top entry B of Blocks
            if B has entries on Inner
                then switch Inner, Outer entries by swopping pointers
            if B still in conflict
                then Halt because non-planar

{ Now we test not to add duplicate entries onto stacks }
        if not Duplicate entry
            then Add Next onto Inner
```

```
{ We add the merged block }
        Add new block onto Blocks corresponding to P and deleted blocks
{ Recursively call Algorithm 2.6 to test planarity of fragment and cycle }
        Recursive_Test_Planarity ( Next, Current)

    end { end of while incident edges unscanned}

{ at this stage all edges incident with Current are scanned }
{ so we backtrack along cycle to parent }

    Current = Parent(Current)

end { end of while not backtrack to root }
```

end

To test that we are not duplicating entries on the stacks, we merely test if the last vertex on the first path from Root is not Next, then we may add Next as an attachment. Of course, if this path is the first path, then we automatically add Next as an attachment. In Algorithm 2.6, below, we present the rest of the algorithm - the special handling of the recursive calls.

Algorithm 2.6: Recursive_Test_Planarity

```
{ Special code for the recursive call from Algorithm 2.5 }

{ Recursively call Algorithm 2.5 to test planarity of fragment and cycle }
    Add end of stack marker to Outer
    Test_Planarity ( Next, Current)
{ Now place all new blocks inside the spine cycle }
    For each new Block
{ check if all fragments may be placed inside spine cycle }
        if entry on Inner and Outer
            then Halt because non-planar
        if entry on Outer
            then move to Inner and delete Block entry
    Delete end of Stack marker
    Add a single new block to represent all new blocks
    Merge the new recursive block with the top block
```

end

Theorem 2.3: Algorithms 2.5 and 2.6 correctly test planarity of a 2-connected graph G, order p and size q, with $q \leq 3p - 6$.

Proof: Lemma 2.2 proves that the initial cycle is found correctly. At all times during the execution of Algorithm 2.5, the stacks Inner and Outer

store the attachments placed inside and outside the cycle respectively. Lemma 2.8 is used to test if we may place the current path on the inside or outside of the Cycle. Lemma 2.9 is used to maintain the stack Blocks correctly. Thus the stack Blocks contains the correct information about the fragment blocks. By trivial induction, the placement of one fragment in a fragment block completely determines the placement of all other fragments in the same fragment block. Also, the placement of one fragment block does not interfere with the placement of any other.
The recursive step in Algorithm 2.6 does not interfere with old entries on Inner and Outer, since Outer has an end of stack marker, and all entries on Inner are less than or equal to Current at the start of the recursive call. At the end of the recursive call, the new attachment information is exactly necessary to continue the operation (i.e. the only attachments left are those less than Current). Thus Algorithms 2.5 and 2.6 correctly test for planarity. ❑

Theorem 2.4: The complexity of the Hopcroft and Tarjan Fragment Addition Algorithm if O(p).
Proof: The DFS is O(p), as is the sorting pass. The total number of entries in Inner and Outer are bounded by the number of back edges, thus is O(p). Lastly, note that each section switch reduces the number of blocks by one, thus we may bound the number of steps involved in section switching required to O(p), since we note that a section switch involves only changing four pointers. ❑

Section 2.3
The Lempel, Even and Cederbaum Vertex Addition Algorithm

In this section we follow the approach of Even's book [Eve79]. Before we consider the algorithm, we associate a numbering with every vertex, called an st-numbering. Given any edge e = st of a 2-connected graph G, a 1-1 function f : V $\rightarrow$ {1, 2, ..., p} is called an *st-numbering* of G if

(a) f(s) = 1
(b) f(t) = p
(c) For every vertex v $\in$ V(G) - {s, t} there exists vertices u and w adjacent with v such that f(u) < f(v) < f(w).

Lempel, Even and Cederbaum [LEC67] first defined such a function and showed that such a function always exists for G. They gave an algorithm which computes an st-numbering for a graph G where e = st is any edge of G. We describe here a linear algorithm by Even and Tarjan [ET76] which generates an st-numbering (we defer proof of the validity of this algorithm until after we have described the algorithm).

The first stage of this algorithm is a Depth-First search (DFS) whose first vertex is t and whose first edge is t-s (i.e. dfi(t) = 1 and dfi(s) = 2). As usual, the DFS generates a DFS tree T from G. During the DFS we calculate for each vertex v $\in$ V(G), the first lowpoint, $L_1(v)$, and Parent(v). We mark vertices s, t and the edge e = st old. All other edges and vertices of G are marked new.

Next we describe a path-finding algorithm, Algorithm 2.7, which is used in the st-numbering algorithm. Given an old vertex v of T, the algorithm finds a directed path P which begins (or ends) at v and proceeds along new vertices and edges to an old vertex. The input to Algorithm 2.7 is the current vertex v, the output is a path P (which could be empty), starting or ending at v. All edges and internal vertices are new.

Algorithm 2.7 : Path_Finding (v, P)

{ Returns a path P starting from vertex v }

There are four possible cases (we choose the first case that is satisfied)

(i) If there is a new back edge v $\xrightarrow{e}$ w then Mark e old.
Output P : v $\xrightarrow{e}$ w.

(ii) If there is a new tree edge v $\xrightarrow{e}$ w then trace a path P whose first edge is e and follows the path defined by $L_1(w)$. That is, the path consists of a series of tree edges ending in a back edge into a vertex u such that dfi(u) = $L_1(w)$. All vertices and edges in P are marked old. Output P.

(iii) Suppose there is a new back edge w $\xrightarrow{e}$ v into v then: Start the path P with e (going backwards along e) proceed along tree edges by using parent values, until an old vertex is encountered. All vertices and edges in P are marked old. Output P.

(iv) The final case is when there is no edge e incident with v which is marked new. Output an empty path P.

end. {of algorithm Path_Finding}

Lemma 2.10: If the path-finding algorithm is always applied from an old vertex v ≠ t, then all ancestors of an old vertex are also old.

Proof: We proceed by induction on the number of applications of the Algorithm 2.7. Clearly, in the initial case (no applications of Algorithm 2.7) the only ancestor of s is t, and t is old. Assume the statement is true up to the present application. We consider each case in Algorithm 2.7 in turn. The first case is trivial, since there are no new vertices marked old; and the lemma still holds. In the second case, since we follow a path of tree edges and we started from an old vertex, all ancestors of those new vertices visited must be old. Recall from Chapter 1, since G is 2-connected, that the back edge reaches an ancestor of the start vertex v, and so the lemma still holds. In the third case we use the same result from Chapter 1 : a back edge must come from a descendant of v. Thus, by following tree edges back up the tree from this descendant we must eventually reach an old vertex (which may be v) and we have the exact same situation as in the second case, where we start from an old vertex and proceed along tree edges and end in a back edge, except now the back edge reaches our current vertex v. So, by the inductive hypothesis, those new vertices now marked old, have all their ancestors marked old. The last case is trivially true and so by induction the lemma is proved. ❑

Corollary 2.2: If G is 2-connected, then each application of the path-finding algorithm produces a path P from an old vertex v of G via new vertices and edges to an old vertex of G, unless there are no new edges incident with v in which case the path produced is empty.

Proof: We consider each of the four cases in Algorithm 2.7. Since a back edge $v \xrightarrow{e} w$ leads from a descendant to an ancestor (Chapter 1), case (i) follows naturally from Lemma 2.10. Similarly, because G is 2-connected, and under Lemma 2.10 again, our result follows for case (ii). Case (iii) follows straight from its definition and case (iv) is trivial. ❑

We are now ready to present the st-numbering algorithm. We use a stack S which initially contains only s and t, where s is on top of t. The vertices s and t are marked old, as is the edge between them. The top-of-stack element is denoted by TOS.

Algorithm 2.8: Generate_st-numbering

{ Generate an st-numbering of a 2-connected graph G }

```
ST-Num = 1
while S ≠ ∅ do          { while elements still need to be numbered }
    Remove the TOS element v from S.
    If v = t
        then        { finished }
            f(t) = ST-Num
            Output f
        else
            Use Path_Finding with vertex v to generate a path P.
            If P is empty    { finished with this vertex }
                then
                    f(v) = ST-Num
                    ST-Num = ST-Num + 1
                else        { add path and vertex onto stack }
                    Let P be v - u1 - u2 - ...- uf- w
                    Push the following vertices onto the stack:
                        uf, uf-1, ..., u2, u1, v
                    in that order so v is still the TOS element)
end
```

Theorem 2.5: For a 2-connected graph G, Algorithm 2.8 computes an st-numbering of the vertices of G.

Proof: We note three points about Algorithm 2.8 :

(i) No vertex ever appears in two or more places on S at the same time. This follows immediately from Corollary 2.2 and the fact that we always add new vertices (i.e. that have just been marked old) first, and the only old vertex added back is the top-of-stack vertex.

(ii) Once a vertex v is on S, nothing under v on S is assigned a number until v has been assigned a number. This follows from (i) and the fact that we push v back onto S after adding each new vertex of a (non-trivial) path which is generated from v onto S.

(iii) A vertex is permanently removed from S only when all incident edges are old.

Next we show that every vertex $v \in V(G) - \{t\}$ is placed on S before t is removed. Since s is placed on stack S initially, we consider a vertex $v \neq \{s, t\}$. Assume that this is not the case. Then, v was not placed on S. Since G is 2-connected, there is a path from s to v which does not pass through t. Let the path be $s = u_1, u_2, u_3, ..., u_\ell = v$. Let u_m be the first vertex on the path which has not been placed on S. But, from point (iii) above, we see that u_{m-1} is only removed from S once all incident edges have been marked old. This implies that u_m must have been on a path from u_{m-1} to another vertex. However, then u_m would have been placed on the stack. This leads to a contradiction. Thus every vertex $v \in V(G)$ is placed on S.

Now we show that Algorithm 2.8 produces an st-numbering. We note from point (ii) that f(s) = 1. Since each vertex v is placed on S and v is eventually removed from S, each v gets an st-number f(v). Now, note that, by Corollary 2.2, each vertex $v \in V(G) - \{s, t\}$ is only placed on the stack as an internal vertex on a new path P. Thus there is an adjacent vertex stored above and below v on S on P. By point number (iii) v's adjacent neighbours will receive the required labellings. The one above v on S will get a lower st-number, and the one below v on S will get a higher st-number. Lastly, vertex t is only removed after each vertex has been removed from S. Thus f(t) = p. ❑

Theorem 2.6: The complexity of Algorithm 2.8 is O(q).

Proof: Firstly, the implicit work of the DFS discussed before Algorithm 2.7 has complexity O(q). Now, consider Algorithm 2.8. During all the calls to Algorithm 2.7, each edge is traversed exactly once (after that it is marked

old), and thus, Algorithm 2.7 has complexity O(q). Finally, the number of stack insertions is related to the size of the graph G and is exactly q+1, and therefore is bounded by O(q). ❑

We can now turn our attention to the actual algorithm by Lempel, Even and Cederbaum [LEC67]. The implementation of this algorithm in a linear time is fairly long. For this reason, we will first consider the algorithm independent of implementation data structures or algorithms.

We assume G is a 2-connected planar graph, whose vertices have been assigned st-numbers. From now on, we will refer to vertices by their st-number. Again, we follow Even [Eve79] for details. First, we direct the edges of G from lower st-numbered vertices to higher to form a digraph D. We define a *graphical source* and a *graphical sink* of a digraph D to be vertices with zero in-degree and zero out-degree, respectively. For st-numbered graphs, by definition there is only one source, namely vertex 1 (vertex s in G), and only one sink (vertex t in G). Consider the digraph $D_k(p_k, q_k)$ defined as having vertex set $V(D_k)$ the vertices with st-numbers {1, 2, ..., k} and arc set E_k consisting of all arcs of D having both end points in V_k. If G is planar, then let $\hat{D}_k$ be a plane realisation of D_k. The following lemma reveals the reason for st-numbering.

Lemma 2.11: Let $\hat{D}$ be a plane realisation of D and $\hat{D}_k$ the plane realisation of D_k in $\hat{D}$. Then, $D - V(D_k)$ lies entirely in one region of $\hat{D}_k$.
Proof: Suppose that this was not the case and we had $D - V(D_k)$ lying in more than one region of $\hat{D}_k$. Then we would have at least two sinks in the digraph D, since no arcs lead from $D - V(D_k)$ to D_k. ❑

Thus, Lemma 2.11 suggests that a planarity testing algorithm would involve building the digraphs D_k with plane realisation $\hat{D}_k$, and adding the vertex numbered k+1 and edges from vertex k+1 to $\hat{D}_k$ to build the digraph $\hat{D}_{k+1}$. If we are unable to construct the digraph D_{k+1} because of unavoidable edge crossings, then G is non-planar.

We now construct a graph B_k from the digraphs D_k and D. We begin with the underlying graph of D_k. For each arc a from a vertex x of D_k to a vertex y in D - $V(D_k)$ we introduce a new vertex (called a *virtual vertex*) and an edge (called a *virtual edge*) which joins x and the vertex corresponding to y. This new vertex receives the same st-number as the vertex of D - $V(D_k)$ to which it corresponds. Note that two distinct (virtual) vertices of B_k may well receive the same st-number. Consider as an example the digraph D and a realisation $\hat{B}_3$ of B_3 shown in Figure 2.10, below.

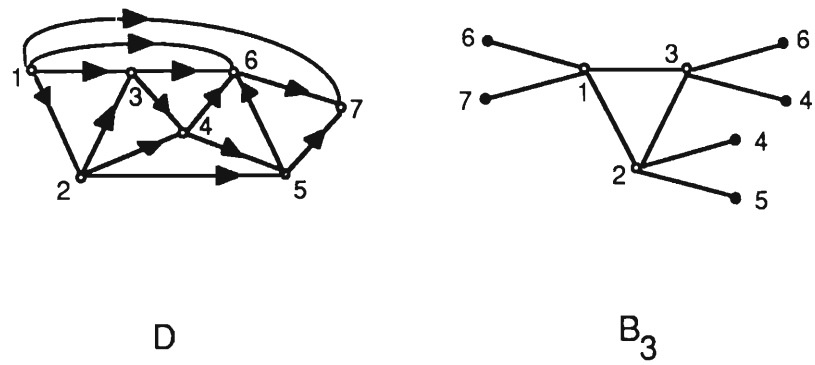


Figure 2.10: Digraph D and $\hat{B}_3$

Now, consider the plane realisation $\hat{B}_k$ of B_k such that the virtual vertices and edges are on the outside region. Since $1 \rightarrow p$ is an arc of D, $1 \rightarrow p$ corresponds to an edge of B_k, we may draw B_k in the following way. Place vertex 1 on the bottom of the diagram. Each vertex i ($1 \leq i \leq k$) is drawn on a separate line of the diagram. The virtual vertices are drawn above vertex k in a straight line called the *frontier* of B_k. This realisation $\hat{B}_k$ is called a *bush form* of B_k, and the planarity algorithm we are to consider relies heavily on the properties of the bush forms $\hat{B}_k$. Consider Figure 2.11 showing bush form $\hat{B}_3$ of B_3 shown in Figure 2.10.

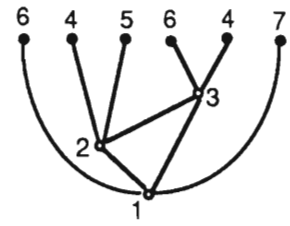


Figure 2.11: Bush form $\hat{B}_3$ of D

Now, using this representation, Lemma 2.11 implies that a bush form $\hat{B}_k$ of B_k exists such that all virtual vertices numbered k+1 appear consecutively in the frontier. If this were false, then there is no realisation of B_k such that the edges from D_k to vertex k+1 do not cross any other edges going to D - $V(D_k)$. Then we would have two regions of D_{k+1} containing vertices of D - $V(D_{k+1})$ and D would be non-planar. The algorithm therefore reduces to obtaining a plane realisation of B_k such that all virtual vertices labelled k+1 appear consecutively across the frontier of B_k. Then we merge the vertices labelled k+1 into one vertex, "pull" it down onto a new line above vertex k and below the frontier. We add the new virtual edges from vertex k+1 to D - $V(D_{k+1})$ and the corresponding virtual vertices, to form $\hat{B}_{k+1}$. We need to be able to guarantee that we can obtain the correct plane realisation $\hat{B}_{k_1}$ say, of B_k from our constructed realisation $\hat{B}_k$ of B_k. We show now however that through a series of simple transformations we may obtain the correct realisation $\hat{B}_{k_1}$. The following definition will prove useful. If v is a cut vertex of a graph H, then let the components of H - v be $G_1, G_2, ..., G_n$. The graphs $H_i = \langle V(G_i) \cup \{v\} \rangle$ i = 1, 2, ..., n are called *v-blocks* of H with respect to v.

Lemma 2.12: Assume v is a cut vertex of B_k. If v > 1, then exactly one v-block of B_k contains the vertices with st-numbering lower than v. (Note that here we are not concerned with the digraph, but the underlying undirected graph of B_k.)

Proof: The st-numbering implies that there is a path in B_k from vertex 1 to every other vertex u ($u < v$) which does not include v. Thus the vertices 1 and u are in the same v-block. ❑

If we consider a graph G then the v-blocks of G may be permuted around v freely. Further, Lemma 2.12 implies that a cut vertex v of B_k is the lowest numbered vertex in each v-block except for the one containing the vertex 1. Each v-block, except for the one which contains 1, is in bush form, and is a *sub-bush with respect to v.* The sub-bushes may be permuted freely around v, and in addition, each sub-bush may be "flipped" in a reflection operation which reverses the order of the frontier of the sub-bush (we take the sub-bush "off" the plane and turn it upside down). Suppose H is a 2-connected subgraph of a bush form, then we define a *sub-bush with respect to* $\mathcal{H}$ to be a sub-bush with respect to a cut-vertex which belongs to H. We may, of course, recursively perform the same operations of permutation and reflection on the cut vertices of the sub-bushes. Figure 2.12 (a) shows a bush form for some graph B_7. Figure 2.12 (b) shows the bush form permuted about vertex 1 and (c) shows Figure 2.12 (a) reflected about vertex 2. Lastly, Figure 2.12 (d) shows Figure 2.12 (a) reflected about vertex 1.

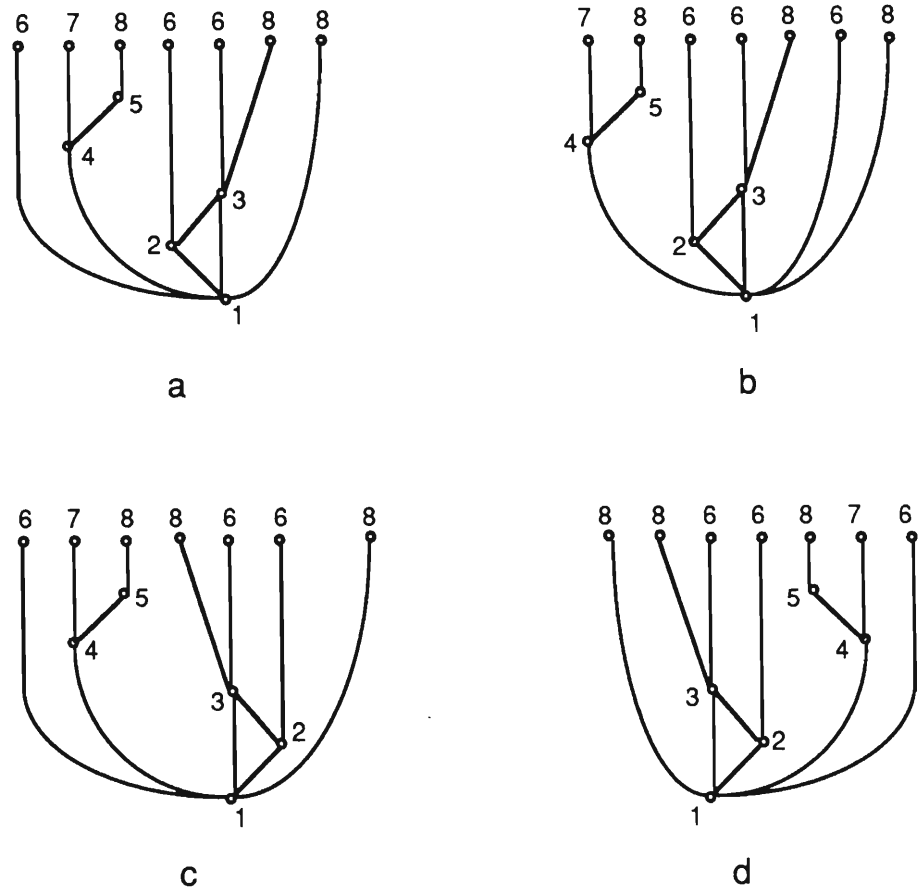


Figure 2.12 (a), (b), (c) and (d) - various Bush Forms

We now need to show that given two bush forms $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$ of B_k, we may obtain $\hat{B}_{k_1}$ from $\hat{B}_{k_2}$ by a simple series of permutations and reflections. But first we need a lemma.

Lemma 2.13: Let H be a block of B_k and $y_1, y_2, ..., y_n$ the vertices of H which are also end points of edges of B_k - V(H). Let $\hat{H}$ denote the plane realisation of H in a bush form $\hat{B}_k$ of B_k. Then, in every bush form $\hat{B}_k$ of B_k, the vertices $y_1, y_2, ..., y_n$ appear on the boundary of the outside region of $\hat{H}$, and in the same order, except for possibly a reversal in their order.
Proof: Since B_k is a bush form, $y_1, y_2, ..., y_n$ will appear on the boundary of the outside region of $\hat{B}_k$. Let $\hat{B}_{k_1}$ be a bush form which has a realisation $\hat{H}_1$ of H such that $y_1, y_2, ..., y_n$ appear in that order on the boundary of the

outside region of $\hat{H}_1$. Assume to the contrary that there is another bush form $\hat{B}_{k_2}$ which has a realisation $\hat{H}_2$ of H such that y_m and y_n which appear in that order around the boundary of the outside region of $\hat{H}_1$ do not appear in order around the boundary of the outside region of $\hat{H}_2$. But then, there must be two vertices y_i and y_j which appear between y_m and y_n on the boundary of the outside region of $\hat{H}_2$. In $\hat{H}_1$, there is a path from y_i to y_j which does not contain the vertices y_m or y_n. Similarly, there is a path from y_m to y_n which does not contain vertices y_i and y_j (see Figure 2.13). But both these paths, which are totally disjoint, must necessarily be embedded in $\hat{H}_2$, which is impossible if $\hat{H}_2$ is a planar embedding of H with the property that y_m, y_n, y_i and y_j are on the boundary of the exterior region. ❑

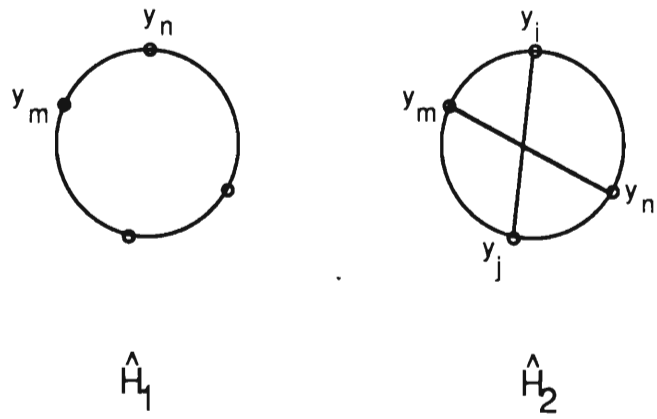


Figure 2.13 : $\hat{\mathcal{H}}_1$ and $\hat{\mathcal{H}}_2$

Theorem 2.7: If $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$ are two bush forms of B_k, then there exists a sequence of permutations and reflections which transform $\hat{B}_{k_1}$ into another bush form $\hat{B}_{k_3}$, say, which has the same frontier as $\hat{B}_{k_2}$.

Proof: We prove it by induction on n, the order of the non-virtual vertices of the bush or sub-bushes. If each bush $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$ has one non-virtual

vertex, then clearly $\hat{B}_{k_1}$ may be transformed through a series of permutations and reflections to have the same frontier as $\hat{B}_{k_2}$.

Suppose n is at least 2, and that the theorem holds for all bush forms with fewer than n non-virtual vertices. Suppose $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$ are two bush forms of B_k which have n non-virtual vertices. Now, let v be the lowest numbered vertex of B_k. If v is a cut vertex, then the v-blocks of B_k appear as sub-bushes with respect to v in $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$. By permuting the sub-bushes with respect to v in $\hat{B}_{k_1}$, we can arrange them to be in the same order as in $\hat{B}_{k_2}$. By the inductive hypothesis we may transform each sub-bush of $\hat{B}_{k_1}$ to have the same frontier as the corresponding sub-bush in $\hat{B}_{k_2}$. So the result follows in this case.

If v is not a cut vertex, then let H be the maximal block to which v belongs. By Lemma 2.13, the sub-bushes with respect to H, appear in the same order (except for possibly a reflection) about the outside region of the two realisations $\hat{H}_1$ and $\hat{H}_2$ of H in $\hat{B}_{k_1}$ and $\hat{B}_{k_2}$, respectively. If they are not in the same order, then reflect $\hat{B}_{k_1}$ about v. They now appear in the same order. By the inductive hypothesis each of these sub-bushes of $\hat{B}_{k_1}$ with respect to H may be transformed via a series of permutations and reflections to a sub-bush whose frontier is the same as the corresponding sub-bush with respect to H in $\hat{B}_{k_2}$.

Thus, in both cases, through repeated application of the above two operations, we obtain a new bush form $\hat{B}_{k_3}$ from $\hat{B}_{k_1}$, which has the same frontier as the bush form $\hat{B}_{k_2}$, and the theorem follows. ❑

Corollary 2.3: If G is planar, then there is a realisation $\hat{B}_k$ of a bush form of B_k, such that the virtual vertices labelled k+1 all appear consecutively in the frontier of $\hat{B}_k$.

In the next section we show how to perform the correct sequence of permutations and reflections to get from our bush form $\hat{B}_k$ to a bush form

$\hat{B}_k$ which has vertices labelled k+1 appearing consecutively. In [LEC67], Lempel, Even and Cederbaum use a form of regular expression parsing to perform the transformations, the complexity of which has not been shown to be linear. We shall restrict our study to a linear algorithm by Booth and Lueker [BL76]. They use a PQ-tree data structure to perform the transformations.

Section 2.4

PQ-trees and a Linear Vertex Addition Algorithm

A PQ-tree is a data structure which has a variety of applications. These include applications to triconnectivity [Kar89], testing the consecutive ones property [BL76] and testing the isomorphism of interval graphs [BL76] and [BL79]. We focus here on their applications to planarity testing, in particular to the planarity testing algorithm of Lempel, Even and Cederbaum [LEC67] presented in Section 2.3.

Let $\mathcal{U} = \{a_1, a_2, ..., a_m\}$ be a universal set. Then the class of *PQ-trees* over the set $\mathcal{U}$ is defined to be the set of all rooted trees whose leaves are elements of $\mathcal{U}$ and whose internal vertices are either '*P-nodes*' or '*Q-nodes*' where a P-node is a node whose children may be permuted freely amongst themselves and a Q-node is a node whose children remain in a fixed order, although the order in which they appear in any planar realisation of the PQ-tree may be reversed. The properties of the P-nodes and Q-nodes are shown in the way we draw them in the PQ-tree. For a P-node, we use a circle which represents freedom of order, whereas a Q-node is represented by a rectangle which portrays the fact that the order amongst the children is fixed. As for the drawing of normal trees, we draw the children of a node below the node in question (thus the root is at the top of the diagram). Figure 2.14 shows an example of a PQ-tree.

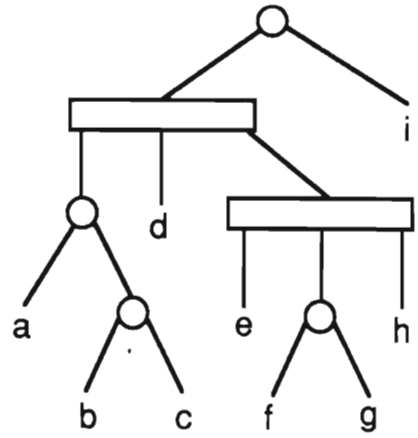


Figure 2.14 : An example PQ-tree

Furthermore, a PQ-tree is called *proper* if we place the following restrictions on P-nodes and Q-nodes :

(i) Every element a_i appears exactly once as a leaf.
(ii) Every P-node has at least two children.
(iii) Every Q-node has at least three children. (This convention is adopted because there is no distinction between P-nodes and Q-nodes with less than three children.)

From now on, except where explicitly specified otherwise, we shall consider only proper PQ-trees. Also, for any PQ-tree T with a realisation $\hat{T}$, when we refer to T we shall also implicitly be referring to the embedding $\hat{T}$. Consider a PQ-tree T. We define the *frontier* of T to be the leaves of T as read from left to right in the embedding. An *equivalence transformation* on a PQ-tree node either :

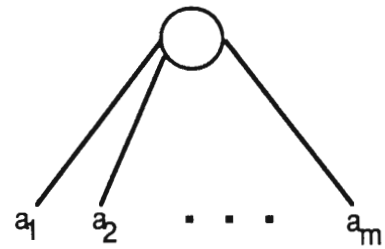
(i) Arbitrarily permutes the children of a P-node; or
(ii) Reverses the children of a Q-node.

We say that two PQ-trees T_1 and T_2 are *equivalent* if and only if the T_1 can be transformed via a series of zero or more equivalence transformations to T_2. If two PQ-trees are equivalent, then we write $T_1 \equiv T_2$. We note that this is an equivalence relation. For example, for the PQ-tree T in Figure 2.14, there are 64 trees in the equivalence class. Every two PQ-trees in the same equivalence class have different frontiers. The set of possible frontiers which we may obtain via equivalence transformations on T is called the set of *consistent permutations* of T, and is denoted by

$$\text{Consistent (T)} = \{\text{Frontier (T')} \mid T' \equiv T\}$$

It is important to note that a PQ-tree T of a universal set $\mathcal{U}$ is a description of the "allowable" permutations of $\mathcal{U}$. The frontier of every PQ-tree $T' \equiv T$ represents a valid permutation. Given a universal set $\mathcal{U}$, there is a spectrum of possible PQ-trees. At the ends of the scale, we have

the *null tree* and the *universal tree.* The null tree is the empty PQ-tree, which is a PQ-tree without any nodes or leaves. Formally, the null tree is not actually a PQ-tree, but we include it for the sake of completeness. The null tree represents the most restricted PQ-tree - one in which we do not know anything about the allowable structure of the universal set $\mathcal{U}$. The set of consistent permutations of the null tree is empty. The universal tree is the most unrestricted PQ-tree - a single P-node with children a_i for all $a_i \in \mathcal{U}$. Thus the universal tree represents every possible permutation on the universal set $\mathcal{U}$. These two trees are shown below in Figure 2.15.



Null tree Universal Tree

Figure 2.15 : The Null tree and Universal tree

Turning to Q-nodes, there are always two *endmost* children. These are the children which are always at the ends of the sequence of children of the Q-node (irrespective of reversal). The rest of the children are *interior.* Furthermore, we say that an *immediate sibling* of a Q-node's child is a child of the Q-node which appears adjacent to that node in the frontier of every PQ-tree T' such that T' $\equiv$ T. Conceptually, the immediate siblings of a node are the neighbours of that node in $\hat{T}$. Thus, every interior node has exactly two immediate siblings and every endmost child has exactly one immediate sibling. The two endmost children will be useful for processing the Q-nodes in the PQ-tree efficiently, as we shall see later. Figure 2.16 illustrates these terms.

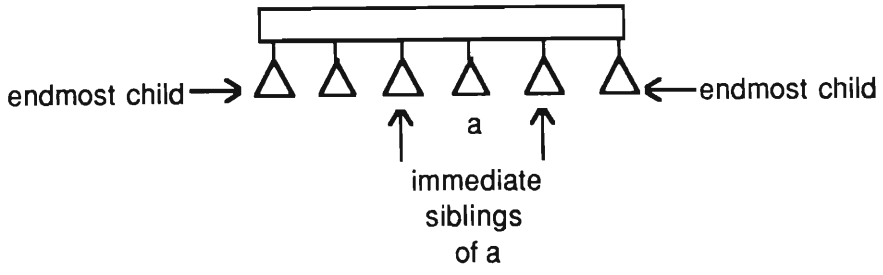


Figure 2.16 : Children of a Q-node

We now discuss the general application of PQ-trees, which we later adapt to perform the planarity testing algorithm of Section 2.3. Let $\mathcal{S}$ be a class of subsets of a universal set $\mathcal{U}$ (note that the elements of $\mathcal{S}$ need not be disjoint). Consider the following problem $\mathcal{P}$: Determine all permutations π on $\mathcal{U}$ so that for every S $\in \mathcal{S}$, the elements of S appear consecutively in π. Algorithm 2.9 gives a formal description of a method which solves $\mathcal{P}$.

Algorithm 2.9: Reduction ($\mathcal{U}$, $\mathcal{S}$)

{ find all permutations π of $\mathcal{U}$ such that, for every set S $\in \mathcal{S}$, all elements of S appear consecutively in π }

Π = {π | $\pi$is a possible permutation of $\mathcal{U}$}
For every S $\in \mathcal{S}$ do
    $\Pi = \Pi \cap$ { π | all objects of S are consecutive within π}
        [the reduction phase]

end

We now describe how the PQ-tree data structure can be used to implement this reduction efficiently. We begin with a universal tree T whose leaves are the elements of $\mathcal{U}$. Note that at this stage Consistent(T) consists of all permutations of the elements of $\mathcal{U}$. The inner loop of Algorithm 2.9 is actually a modification procedure which adjusts T to reflect the new constraint, namely that for each S $\in \mathcal{S}$ the elements of S appear consecutively in the frontier of T. Effectively we are *reducing* the size of Consistent(T) (i.e. the set Π). We say that T is *$\mathcal{S}$-reduced* if, for every T' $\cong$ T, the elements of S appear consecutively in Frontier(T'). We only need a single operation on T, denoted by Reduce (T,S), which modifies T so that it becomes S-reduced. Informally, the process of performing such a reduction consists of scanning the PQ-tree node by

node, starting from the leaves, and then looking for *patterns* in the structure of the subtree at each node, and structurally modifying the subtree rooted at the node with a *replacement*. We call each pair of pattern and replacement at a node a *Template*. We now provide a formal description of this reduction.

Algorithm 2.10: Reduce (T, S)

{ Constrain the PQ-tree T so that S appears consecutively in Frontier(T) }

```
QUEUE = empty
    { Queue contains nodes which can be matched and then replaced }
for each leaf ∈ U do
    Add_to_Queue (leaf)
while not Finished do { Finished when all elements of S consecutive }
    Current = Head of Queue
    Match Templates to Current                          { match a template }
    if a Template matches
        then Replace subtree rooted at Current          { add replacement }
                with replacement pattern
            else                                        { error - no match }
            T = Null Tree
            Halt
    if S ⊆ {Leaf | Leaf is a leaf of the subtree of T rooted at Current }
        then Finished          { we have arranged all leaves of S }
                               { consecutively in Frontier(T) }
        else
            if parent of Current_Node has all its children queued
                then Add parent of Current to Queue
```

end

Notice that the changes are local, we only modify the node and its children. Also, the pattern which is matched depends only on the node and its children. We match the children of a node before we match the node itself. This suggests the usage of a queue; we only add an element to the queue once all its children have been matched. Secondly, this suggests a bubble-up procedure where we process descendants first. We shall prove later that Consistent(Reduce(T,S)) is the subset of Consistent(T) with the property that each of these permutations has the elements of S appearing consecutively.

There are three possible states a child of a node can be in. Consider any child X. If none of the descendants of X which are leaves are in S, we say that X is *empty*. If all of the descendants of X which are leaves are in S, we say X is *full*, and lastly if some of the descendants of X that are leaves are in S, but some are not, then we say X is *partial*. The templates matchings look for a combination of the node type, i.e. whether the node is a P-node or a Q-node or a leaf, and the states of its children. A node is said to *pertinent* if some or all of its children are either full or partial with respect to S. The *pertinent subtree* of T with respect to S, denoted Pertinent(T,S), is the unique subtree, rooted at a vertex X, of T of minimum height whose frontier entirely contains S. The root of the pertinent subtree is denoted by Root(T,S). For example, in Figure 2.17, below, we show the pertinent subtree of the PQ-tree given in Figure 2.14 when S = {a, c, e, f}. Intuitively we should only have to perform operations on Pertinent(T,S).

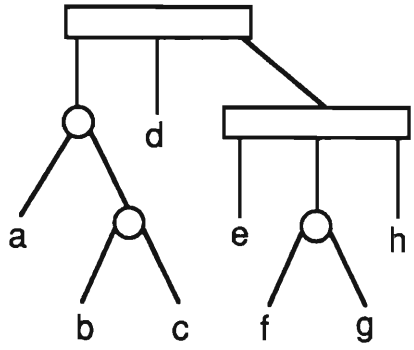


Figure 2.17 Pertinent(T,S) when S = {a, c, e, f}

After matching each template and performing the appropriate replacement, we need to ensure that there is no information loss. In other words, the PQ-tree Reduce(T,S) must represent exactly every possible valid permutation of the frontier of T which has all elements of S appearing consecutively. We shall now describe all possible templates and replacements for a PQ-tree T. The following convention is adopted. Full nodes are shaded, partial nodes have only the right-hand side of the node shaded. Empty nodes are left unshaded. Whenever a child's node type does not matter, we shall represent it by a shaded or unshaded triangle. Furthermore, for this and the next chapter, we shall always arrange the children of a partial node X in T so the pertinent descendants

which are leaves of the maximal subtree rooted at X appear consecutive in Frontier(T). There are a number of templates, and we shall assign them two letter names. We will see next that there are seven templates for P-nodes (named P_0, P_1, ..., P_6), four templates for Q-nodes (named Q_0, Q_1, ..., Q_3) and two leaf templates L_0 and L_1.

Consider the leaves of T. Clearly there are only two templates, either a leaf is not in S, or it is in S. The replacement pattern is simply the same node unchanged except to note that the node is empty or full.

We consider P-nodes and Q-nodes separately. For P-nodes there are seven templates. Firstly, there are the cases where the children are either all empty or all full. Figures 2.18 a and b, below show these cases respectively, giving Templates P_0 and P_1. Note that for Template P_1 the replacement is exactly the same as the original, except that the state of the node is now determined. This way we do not restrict the freedom of elements within S to permute amongst themselves.

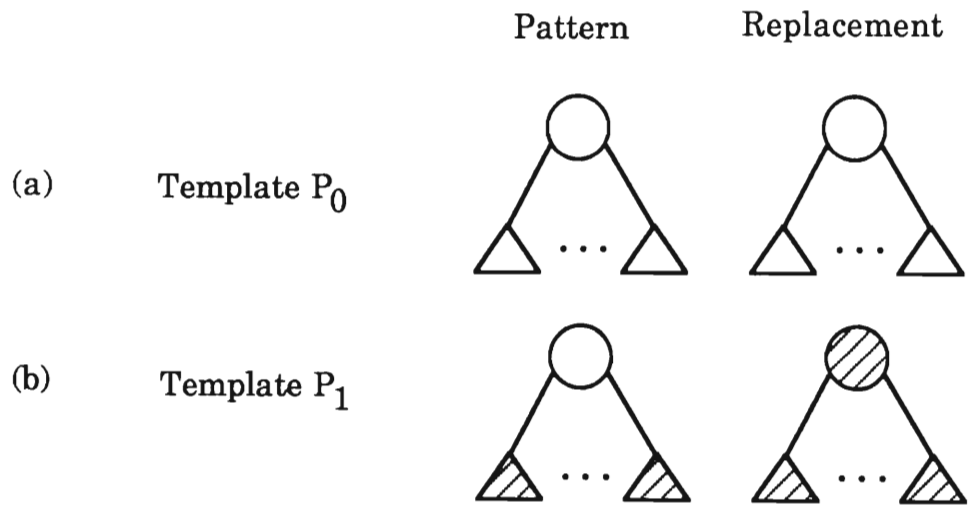


Figure 2.18 : Templates P_0 and P_1

The next case to consider is the one where some of the children are full and some are empty. This case we split into two further cases. When the current node we are at is Root(T,S), then we must cluster all the full children together, but still allow the full children, as a cluster, and the

empty children to permute freely. Figure 2.19 shows Template P_2. Notice that the replacement does not restrict the permutations any more than necessary; we allow the cluster of full nodes to permute, as a cluster, amongst the empty children and for full children to permute freely amongst themselves.

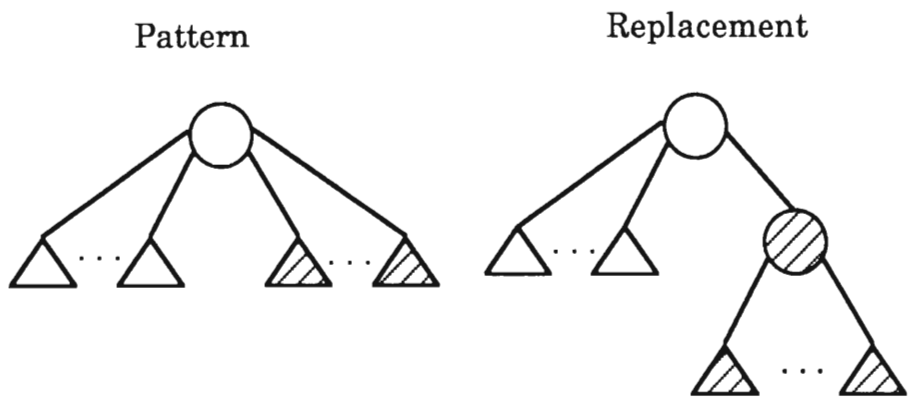


Figure 2.19 : Template P_2 for Root(T,S)

The second sub-case to consider occurs when we have some full children and some empty children at a node which is not Root(T,S). We partition the children into two groups, namely the full children and the empty children, and allow them to permute freely within the same cluster. This is in contrast to Template P_2 where the cluster of full children could permute with the empty children. The same situation does not exist here because we know that there are other elements of S elsewhere in the pertinent tree (or else we would be at Root(T, S) and Template P_2 would be used). A permutation with empty children on either side of the full group of children would not allow the other full children elsewhere in the pertinent tree to be adjacent to these full children. We label the node partial. Figure 2.20 shows this template, Template P_3.

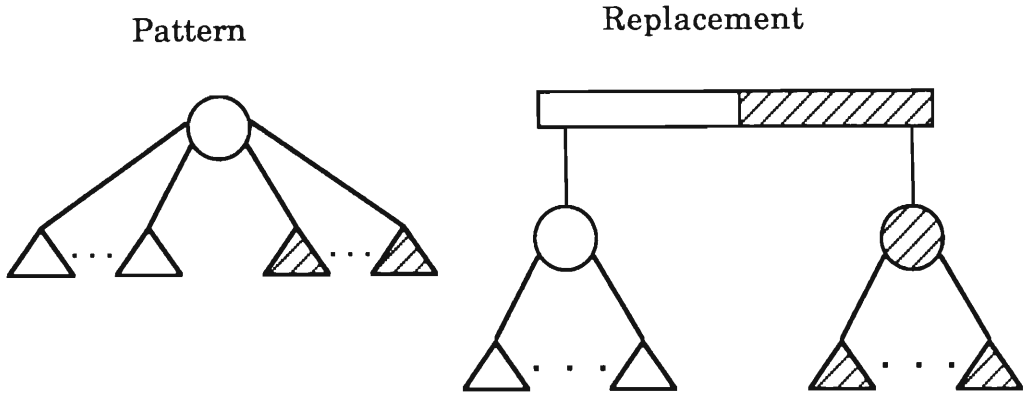


Figure 2.20 : P_3 for partial nodes not Root(T,S)

Notice that we change the node type to a Q-node, this is done to simplify the template definitions later. Although the PQ-tree is no longer proper (since the replacement Q-node only has two children), this situation will be rectified later in the template matching process. In the special cases where there is only one full child or one empty child, Figure 2.21 shows the replacements for the Template P_3.

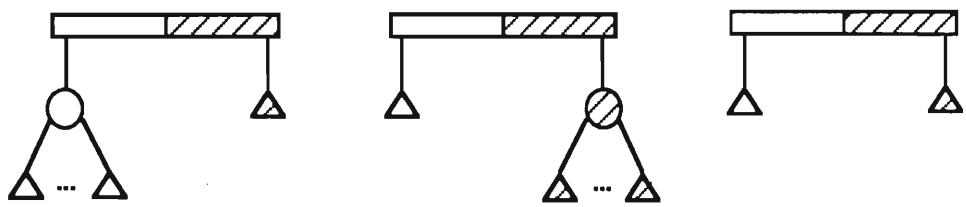


Figure 2.21 : Special Replacement cases for P_3

Now consider the case where at least one of the children is partial. As can be seen from the rest of the templates, only a Q-node may be partial. The first case is a special case where there is exactly one partial child, possibly some full children and possibly also some empty children. Again, this breaks into two special cases, where the node in question is Root(T,S) and where it is not Root(T,S). Consider the first case shown in Figure 2.22.

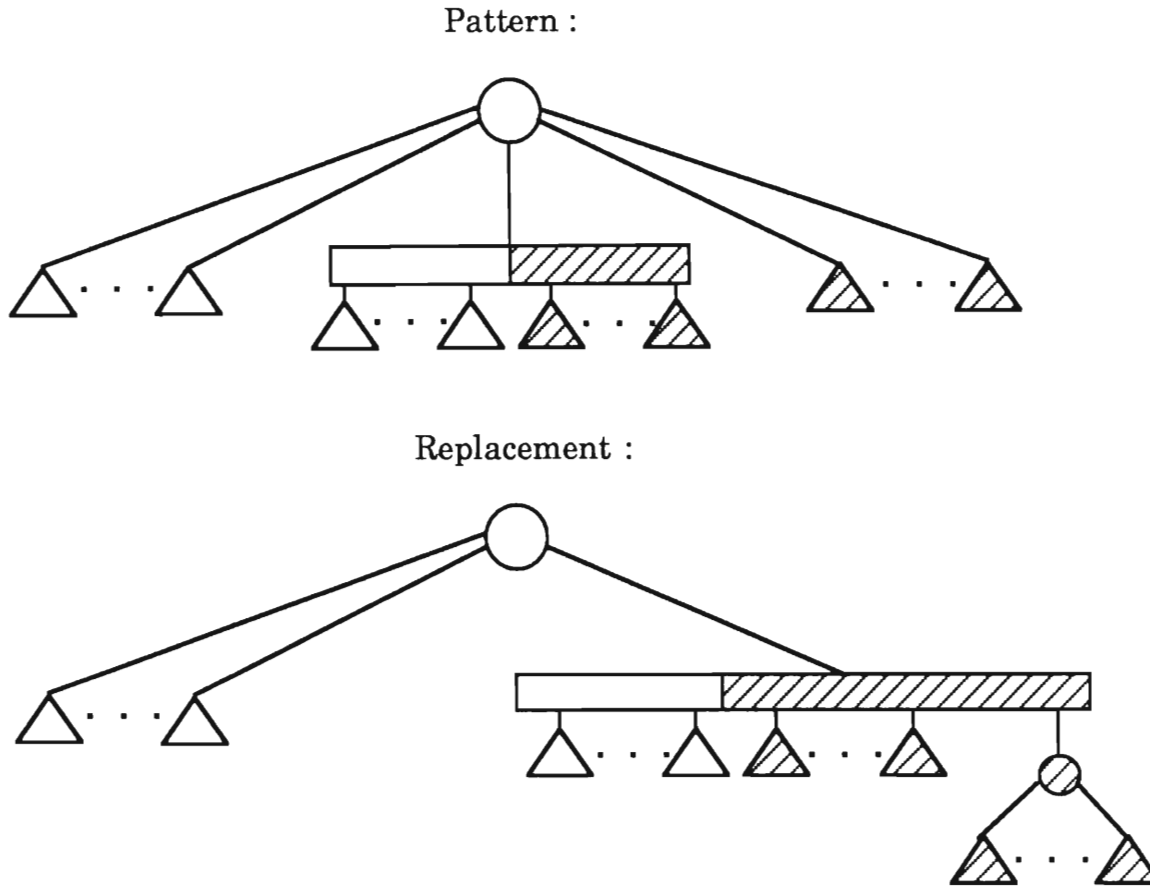


Figure 2.22 : Template P_4 for a node at Root(T,S) with one partial child

Again, as in Template P_2 the full cluster may permute freely amongst the empty children. However, this time we must restrict the full (empty) children which were restricted to a fixed order in the partial Q-node to remain in that order. Note that the full children of the node may permute amongst themselves, but they must always be on one side of the Q-node (because we group all full children together).

The second sub-case is where the node in question is not Root(T,S) and we have exactly one partial child, no or possibly some full children and no or possibly some empty children. The same transformation is performed, except, as for Template P_3, we restrict the empty children of the node to permute freely only on one side of the Q-node. All fixed position empty and full children remain (necessarily) in their fixed positions. Template P_5 is shown below in Figure 2.23.

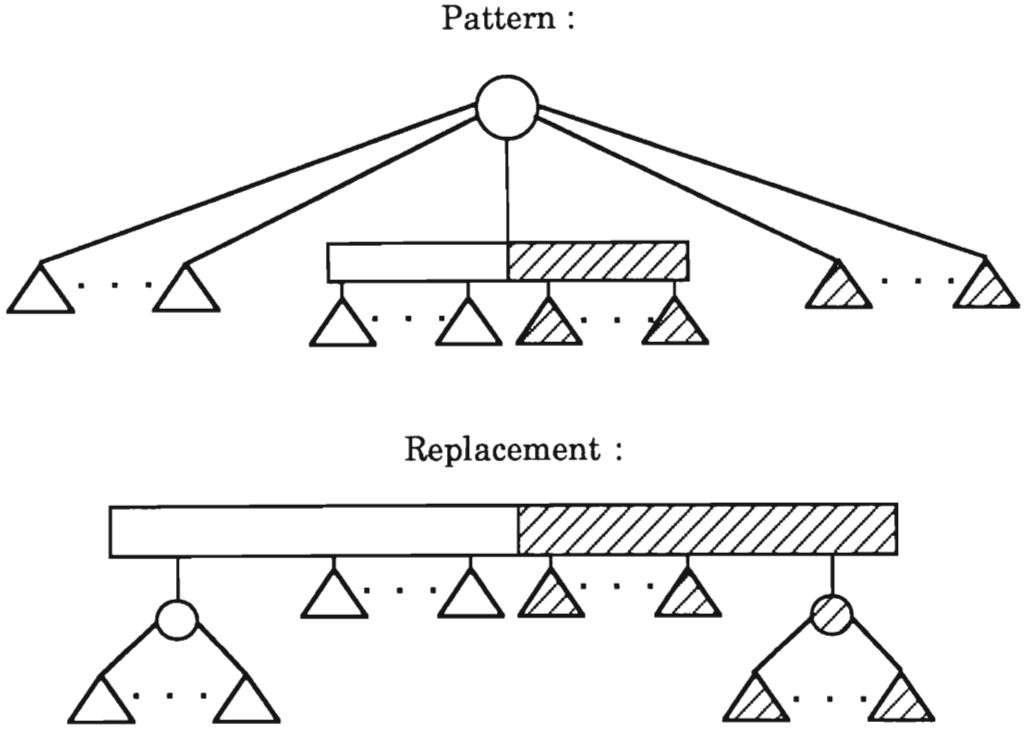


Figure 2.23 : Template P_5 for a node, not Root(T,S) with one partial child

The last case to consider for the P-nodes is when we have exactly two partial children. In this case, the node must be Root(T,S). If it is not, then there is no way we can continue the matching. To see this, consider the Template P_6 shown in Figure 2.24, below. The full children of the node must be placed between the two partial nodes. The full children of the two partial nodes must be adjacent to the other full children. This means that the replacement has empty children on either side of the new Q-node. Hence if this node is not Root(T,S), then we cannot place other full elements from the rest of the pertinent tree adjacent to these full elements.

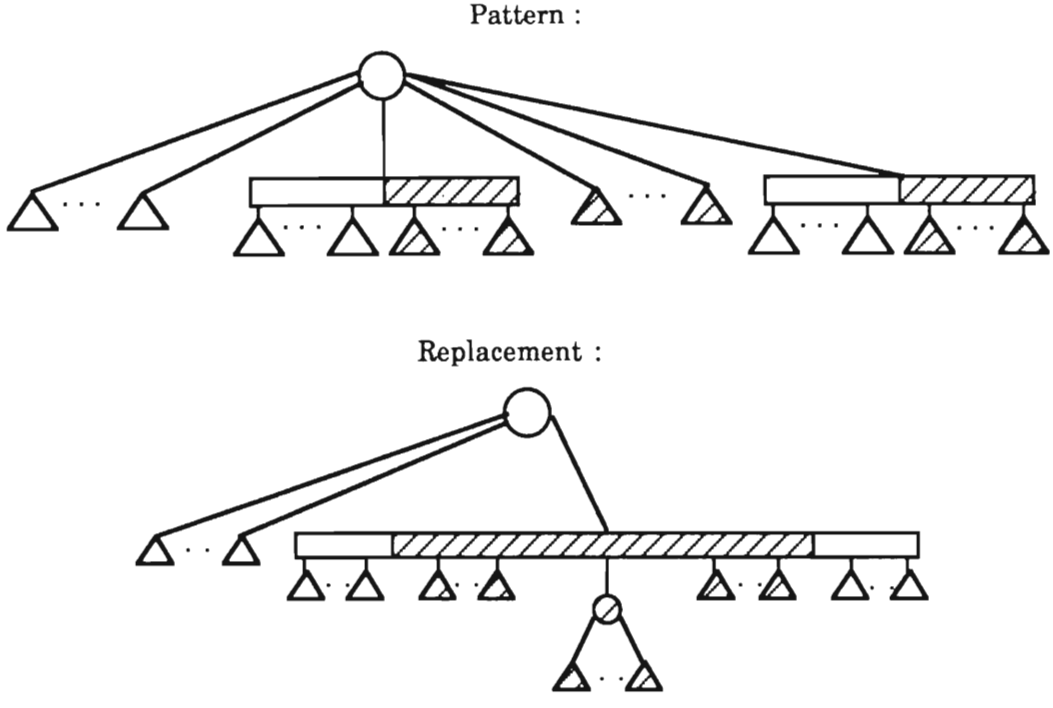


Figure 2.24 : Template P_6 for exactly two partial children

We shall now consider the templates for Q-nodes. The Templates Q_0 and Q_1 are the same as for P-nodes. They cater for Q-nodes whose children are all empty or all full, respectively. Figure 2.25 shows Templates Q_1 and Q_0.

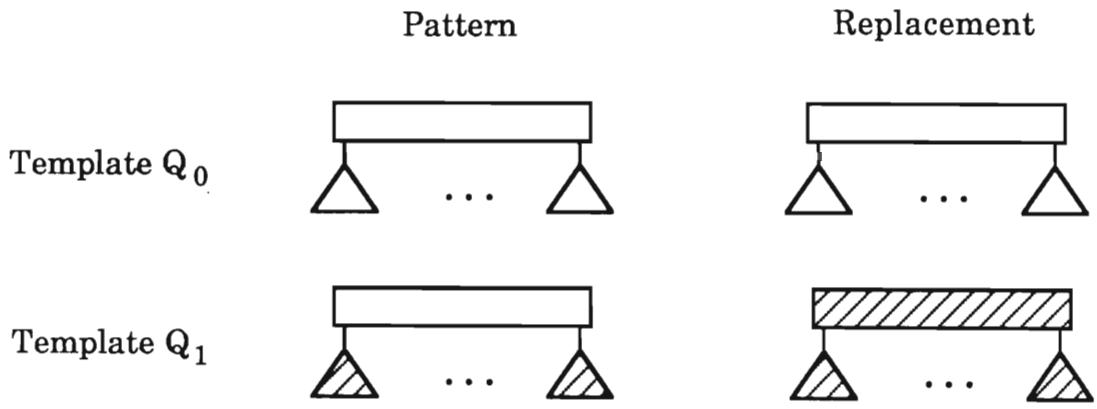


Figure 2.25 : Templates Q_0 and Q_1

Compared to the corresponding P-node cases, matters are much simpler for the rest of the Q-node templates. We can condense the Q-node versions

of Templates P_2, P_3, P_4 and P_5 into one Template Q_2. This template is valid for the pattern shown in the Figure 2.26.

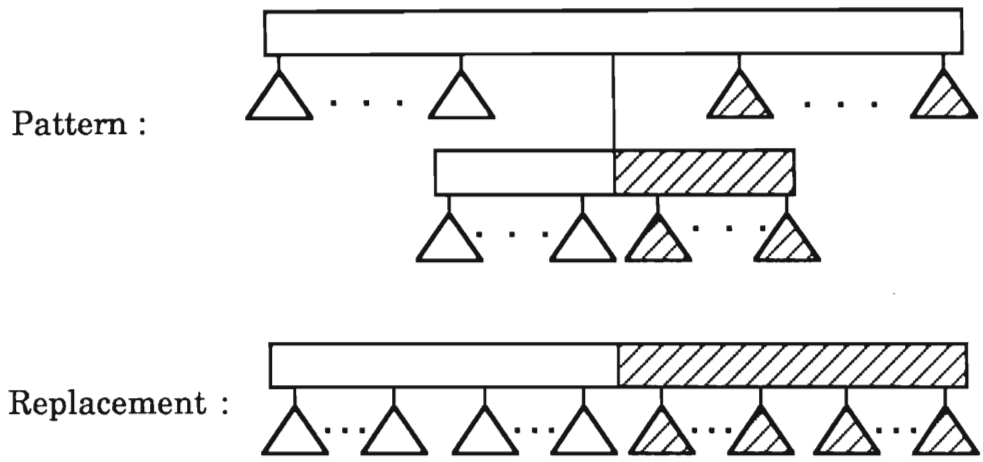


Figure 2.26 : Template Q_2 for at most one partial child

The left to right order of the node's children must be exactly as shown, but some of the information may be missing (we delete the corresponding part from the replacement). There can be up to one partial child, and if present, it must be as shown in Figure 2.26. Either the empty or the full children may be missing from the diagram. If the full children are present, then they must occupy one side of the Q-node in one cluster (i.e. a full child must be an endmost child). The last Q-node template is Template Q_3, and is for the case when the node is Root(T, S) and both endmost children of the node are not full. See Figure 2.27 below.

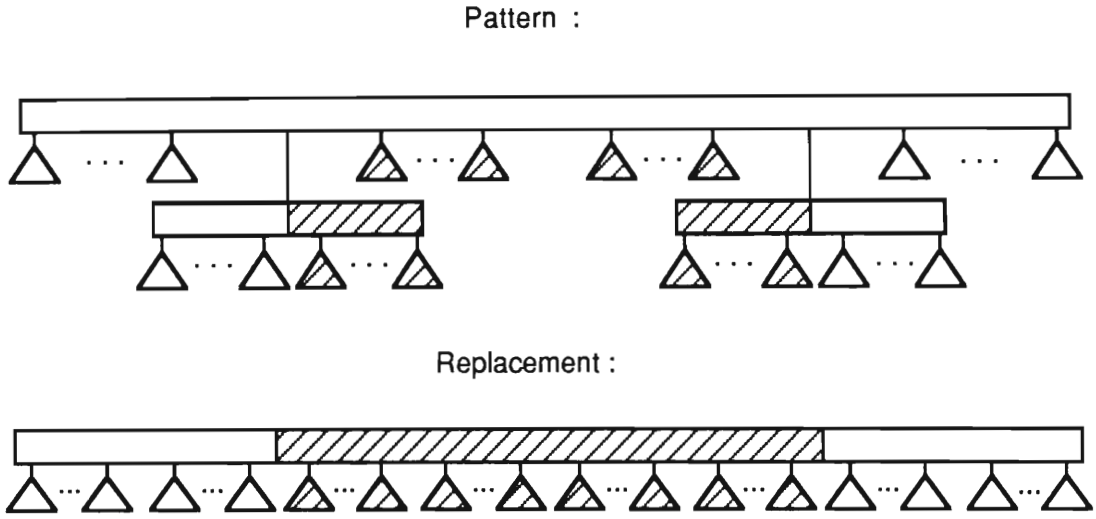


Figure 2.27 : Template Q_3 for Root(T,S) and at most two partial children

Note that this template can only apply to Root(T,S), so exactly the same argument applies as for Template P_6. As in Template Q_2, we may have no partial children but here we can also have up to two partial children. If both partial children are present, then any full children must lie between the two partial nodes. Again, as in Template Q_2, either the full children or the empty children or both may be missing from the template.

Let us consider an example of the template matching process. Unfortunately, it is impossible to construct an example that uses all the templates. For example, Templates P_2, P_4, P_6 and Q_3 are mutually exclusive. Consider as an example the PQ-tree given in Figure 2.28.

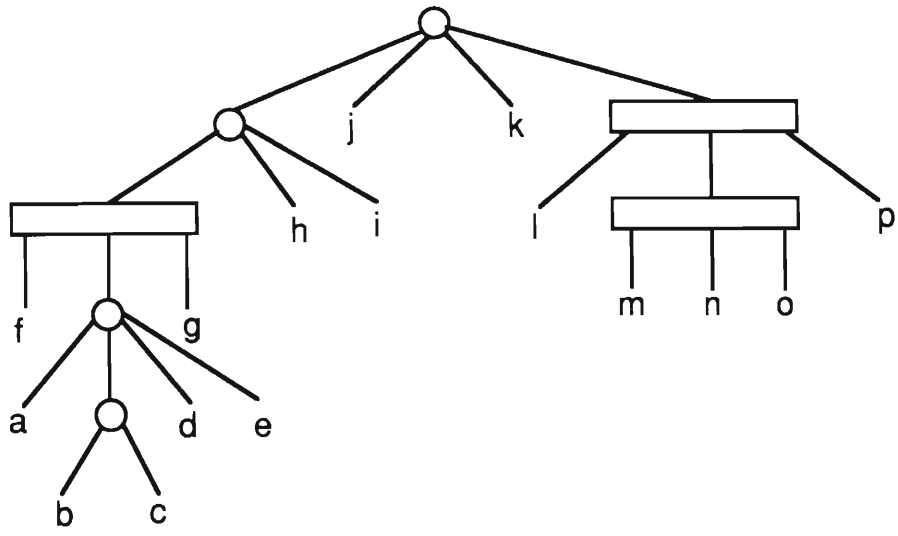


Figure 2.28 : A PQ-tree T

Suppose that S = {b, c, d, f, h, j, m, n, o, p}. Then, the process of template matching will yield the reduced PQ-tree, Reduce(T,S). Clearly, all of the leaf elements {b, c, d, f, h, m, n, o, p} will trigger L_1, and will be labelled full. The rest of the leaves {a, e, g, i, k, l} will be labelled empty. The parent of elements b and c will trigger Template P_1, and the parent of m, n and o will trigger Q_1. Now that the parent of b and c has been labelled, we may match the parent of d to Template P_3, and hence we may match the parent of g to Template Q_2. Similarly, we may now match the parent of p to Template Q_2's pattern. At this stage the partially matched PQ-tree appears as in Figure 2.29.

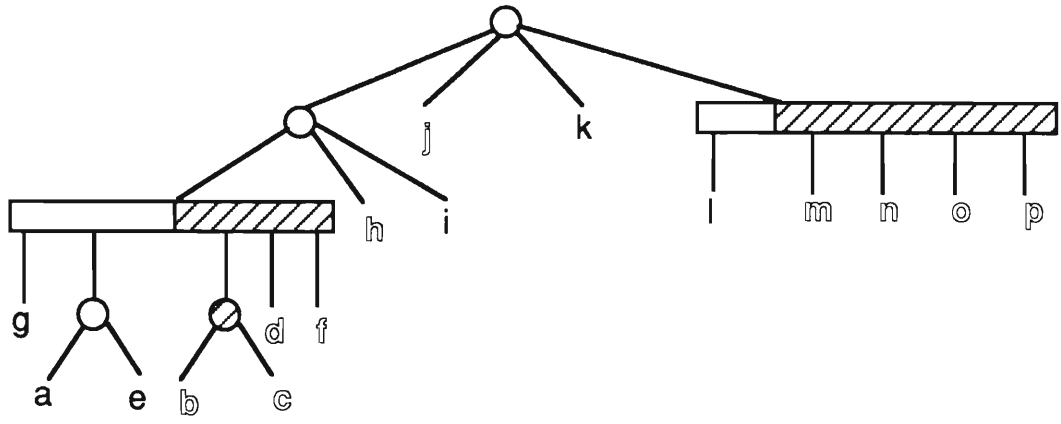


Figure 2.29 : The partially matched PQ-tree T

Now we may match the parent of h with Template P_5. Now that all the children of the root have been matched, we may match Root(T,S) with Template P_6, yielding the final PQ-tree Reduce(T,S) in Figure 2.30.

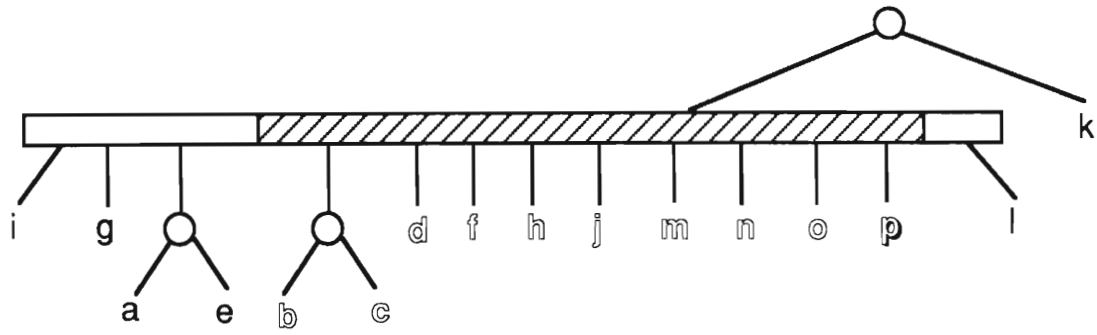


Figure 2.30 : Reduce (T,S) for S = {b, c, d, f, h, j, m, n, o, p}

In Figure 2.30, we can see that Reduce(T,S) has all the elements of S consecutive for every frontier of Consistent (Reduce(T,S)). All that remains now is to prove that the reduction process does indeed only restrict the frontiers of T to those which have the elements of S appearing as a consecutive subsequence in Frontier(T). Given a universal set $\mathcal{U} = \{a_1, a_2, ..., a_n\}$ and a subset $S \subseteq U$, where $S = \{a_k = a_{s_1}, a_{s_2}, ..., a_{s_\ell} = a_m\}$, denote by Consistent($\mathcal{U}$, S) the equivalence class, with respect to PQ-trees, of the PQ-tree shown Figure 2.31.

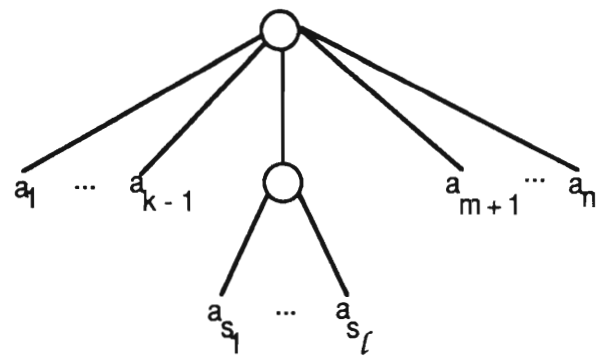


Figure 2.31 : The tree of Consistent(U,S)

Essentially, this PQ-tree gives all possible permutations of $\mathcal{U}$ where elements of the set S appear consecutively in the frontier.

Theorem 2.8: Consistent(Reduce(T,S)) = Consistent(T) $\cap$ Consistent($\mathcal{U}$,S), that is, the set of frontiers consistent with Frontier(Reduce(T,S)) is exactly the set of those frontiers consistent with Frontier(T) which have the elements of S appearing consecutively.

Proof: We prove that there is mutual containment of the two sets. First we prove that Consistent(Reduce(T,S)) $\subseteq$ Consistent(T) $\cap$ Consistent($\mathcal{U}$,S). Take any permutation π from the set Consistent(Reduce(T,S)), and let T' be a PQ-tree from the equivalence class of Reduce(T,S) with frontier π. Since we have a permutation π, such a PQ-tree T' must exist. Now, undo the reduction process in the following manner. Consider that at every stage we matched a pattern, possibly performing some equivalence transformation, and substituted the appropriate replacement. To undo the reduction phase, we merely undo the replacement. We do not undo the equivalence transformation which the particular node underwent to allow the match to be made. After we have undone the reduction phase, we are left with a PQ-tree T'' which has the elements of S appearing consecutively, and which is part of the set Consistent (T) [that T' is part of Consistent(T) is easy to see if we consider that by, where necessary, performing the equivalence transformations (by merely permuting children of P-nodes and reflecting children of Q-nodes) we obtain T from T']. Thus $\pi \in$ Consistent(T). To show that $\pi \in$ Consistent($\mathcal{U}$, S), consider the final template matching of the reduction. After the reduction, for the last template to have been applied, we must have a node Root(T,S) which is either

(i) a P-node with all its descendants in S, or
(ii) a Q-node with all full nodes appearing consecutively (and hence all elements of S appearing consecutively).

Thus, π must be a permutation of $\mathcal{U}$ with the elements of S appearing consecutively, so $\pi \in$ Consistent($\mathcal{U}$, S).

Conversely, take any permutation $\pi \in$ Consistent(T) $\cap$ Consistent($\mathcal{U}$, S). We may find a PQ-tree T' equivalent to T such that Frontier(T') = π. Then, consider the process of template matching in exactly the same order as

we applied the templates to T. This time it is not necessary to apply any equivalence transformations to T'. Since S is consecutive in Frontier(T'), we find that the patterns match without the need of equivalence transformations. Every node, except possibly Root(T',S), has at most one partial node. Every partial node becomes a Q-node, and the sequence of children for each such node will be a sequence of full (empty) children followed by a sequence of empty (full) children. So every node will have a template match. Thus we obtain a PQ-tree T'', but this PQ-tree is in the equivalence class of PQ-tree Reduce(T,S), since we can obtain T from T'' by a simple series of equivalence transformations. So we have $\pi \in$ Consistent(Reduce(T,S)). ❑

Theorem 2.8 is a very important theorem. It states that given a PQ-tree T, we can build these constraints, of each set S of a class $\mathcal{S}$, having its elements appearing consecutively within Frontier (T), into the PQ-tree via a conceptually simple reduction algorithm which is merely a series of template matchings. All that remains of course is to attempt to implement such an algorithm in linear time.

Let us turn to applying the PQ-tree data structure algorithm to the graph planarity testing algorithm of Lempel, Even and Cederbaum [LEC67]. Consider any 2-connected graph G and its st-number labelled digraph D, where each arc is directed from a vertex of lower st-number to a vertex of higher st-number. There is a direct analogy between the PQ-tree reduction of a PQ-tree T (which we shall discuss below) and the successive generation of the bush forms $\hat{B}_k$ with respect to D. Consider the following algorithm using the PQ-tree data structure. We associate edges with elements of our universal set. Start with a single P-node having as leaves children consisting of every edge directed out of vertex v_1. We perform $|p|-1$ reductions on the PQ-tree T. During the k-th reduction, we reduce the PQ-tree with respect to the set S which consists of leaves representing edges which are directed into vertex v_k. This reduction will generate Reduce(T,S), which is T constrained so that the elements of S appear consecutively along Frontier(Reduce(T,S)). Now, remove all the nodes of S and replace them by a single P-node (this is

equivalent to pulling the virtual vertices labelled k down into one vertex in the process of forming $\hat{B}_{k+1}$). Finally, to complete the construction of $\hat{B}_{k+1}$, add children to the P-node which are leaves which represent edges directed from the vertex v_k (which is analogous to adding virtual edges and vertices for edges directed out of v_k thereby obtaining B_{k+1}). The reduction process is then continued. Algorithm 2.11 below details the algorithm.

Algorithm 2.11: Planarity_Testing

{ Test a graph for planarity using PQ-trees }

```
𝒰 = set of edges directed out of v1                    { initial universal set }
T = Universal tree of 𝒰                                { the initial tree }
for Current = 2 to |p|-1 do                            { for each vertex }
    S = set of edges directed into vertex vCurrent     { set to reduce }
    T = Reduce(T,S)                                    { do reduction }
    S' = set of edges directed out of vertex vCurrent  { new leaves }

                                                       { vertex addition step }
    if Root(T,S) is not full
        then Replace sequence of full children and their descendants by a
             P-node with children the elements of the set S'
        else Replace Root(T,S) and its descendants by a P-node with
             children the elements of the set S'
    𝒰 = (𝒰 - S) ∪ S'                                   { update universal set }
```

end

We have omitted details when replacing the full children of a Q-node. We shall return to these details in the next section. We must still check that when we replace the full children of a Q-node by a single P-node that we keep the PQ-tree T proper. That is, we must check after the replacement that the Q-node does not have only two children. If that is the case, then we must convert the node to a P-node. Consider as an example the following graph K_5.

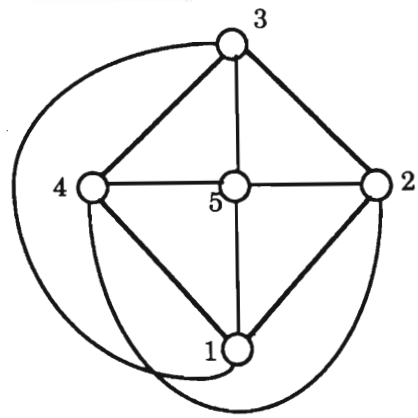


Figure 2.32: K_5

We shall now run the planarity algorithm on $G \cong K_5$. The initial PQ-tree T is shown in Figure 2.33, below. Note how T represents all the edges, yet does not restrict them into any particular combination in a planar realisation of G.

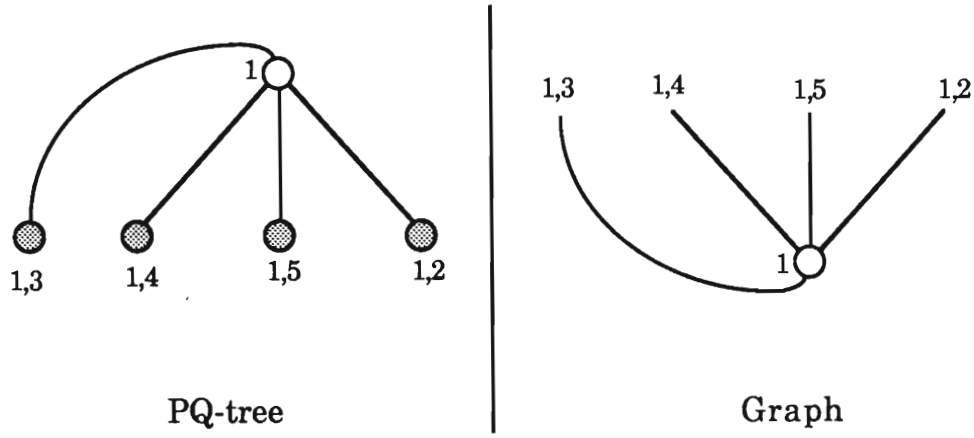


Figure 2.33: Initial PQ-tree T

Now, we gather all leaves representing edges directed into vertex v_2. In this case there is only one such leaf. Thus we replace the leaf (v_1,v_2) with a P-node having children leaves being the edges directed out of vertex v_2, namely the leaves (v_2,v_3), (v_2,v_4) and (v_2,v_5). See Figure 2.34.

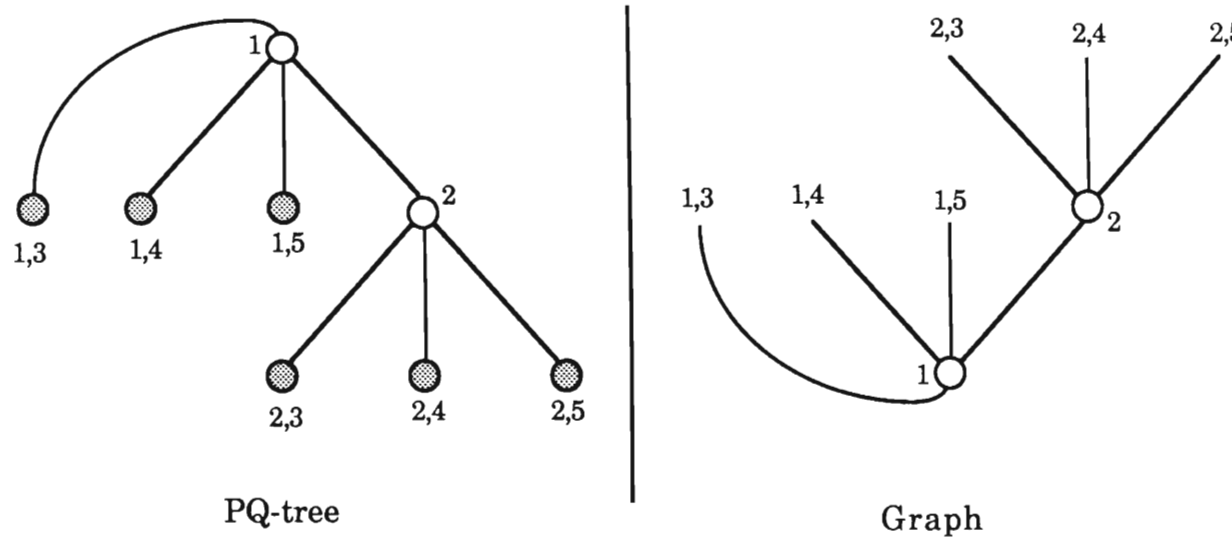


Figure 2.34: After reduction is complete for vertex v_2

Now we consider the set S = {(v_1,v_3), (v_2,v_3)}. This time, the reduction is non-trivial. Template L_1 is triggered twice, once for each leaf element in S. The parent of leaf (v_1,v_3) may not be matched yet, since not all of its pertinent children have been matched. However, we match the parent of (v_2,v_3) to Template P_3. Consequently, we may now match Root(T, S) (which, in this case is Root(T) with Template P_4. Now we remove the sequence of full children from Reduce(T, S) and add in their place a P-node. To this P-node we add the children leaves (v_3,v_4) and (v_3,v_5). The resulting PQ-tree is shown in Figure 2.35.

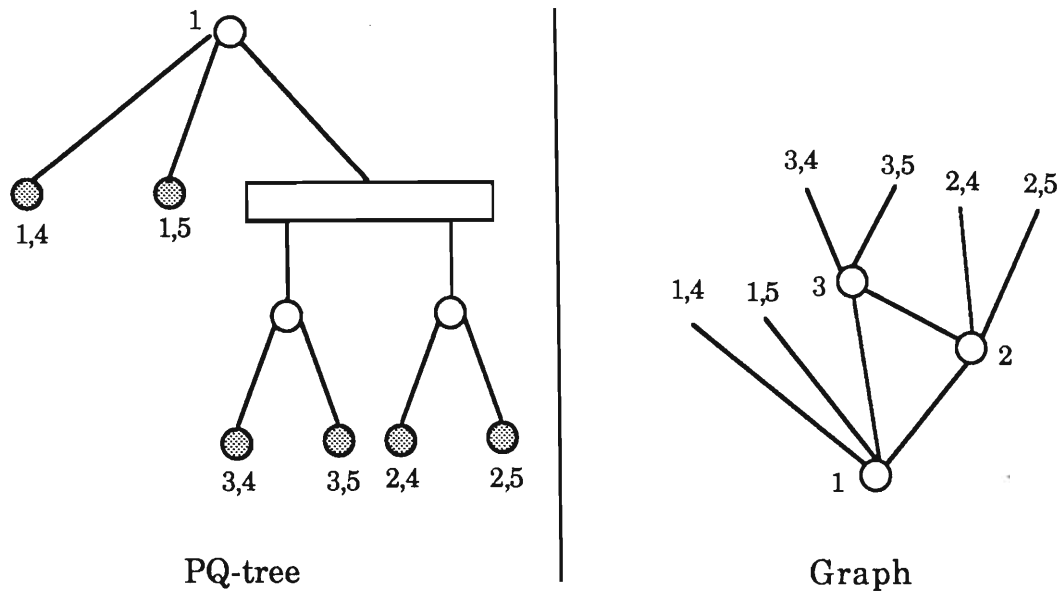


Figure 2.35: After reduction is complete for vertex v_3

Take the next set, namely S = {(v_1,v_4), (v_2,v_4), (v_3,v_4)}. Template L_1 is triggered three times, once for each leaf element in S. The parent of leaf (v_1,v_4) may not be matched yet, since not all of its pertinent children have been matched. However, we match the parent of (v_2,v_4) to Template P_3, and the parent of (v_3,v_4) to Template P_3. At this stage no template matches the next node on the queue. Although Template Q_3 would match, the current node is not Root(T, S) and thus Template Q_3 may not be applied. Consequently we fail the reduction process and conclude that K_5 is non-planar. To see that the reduction process must fail at this stage is easy. Consider Figure 2.36, below, which shows what the PQ-tree T and corresponding sub-graph would look like had we applied Template Q_3 to T. We observe that, in order to group (v_1,v_4) with the other pertinent edges, we would have to cross other edges.

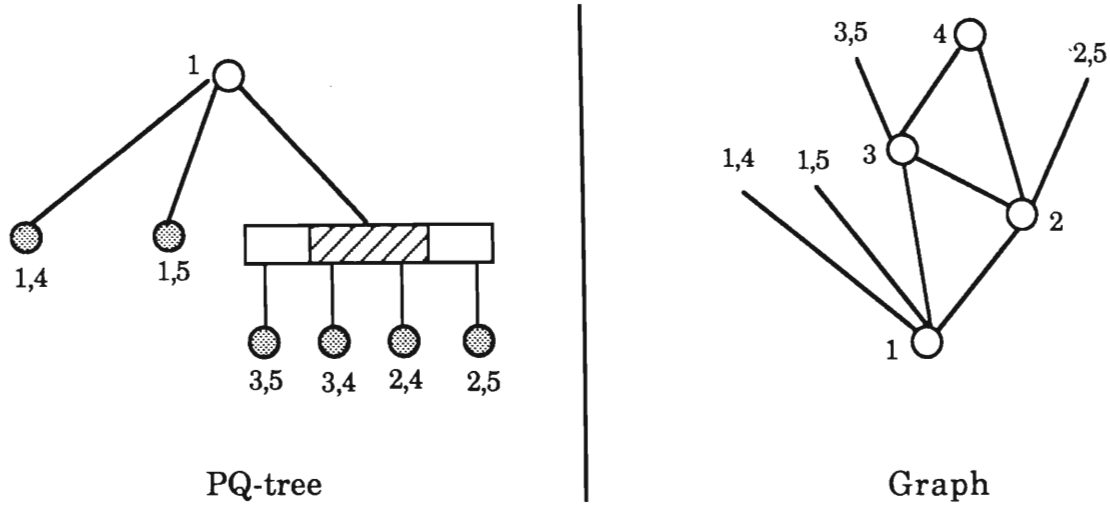


Figure 2.36: PQ-tree after applying Template Q_3 incorrectly

All that remains to be shown, is that there is a direct correspondence between the Bush Forms $\hat{B}_k$, and the PQ-tree T at the end of each reduction process. We say that the PQ-tree T and Bush Form $\hat{B}_k$ are equivalent, if there is a bijective mapping from every $T' \cong T$ to a Bush Form $\hat{B}_k$ of B_k.

Theorem 2.9 : If the reduction procedure, Algorithm 2.11, fails for a 2-connected, st-numbered graph G then G is non-planar.

Proof : We first prove that, at any stage in the algorithm, there is a direct correspondence between the Bush Forms and our modified PQ-tree T. That is, if we can arrange all virtual vertices labelled k+1 of the bush form $\hat{B}_k$ (by reflecting 2-connected components and permuting around cut-vertices) to appear consecutively in the frontier of $\hat{B}_k$, then we can arrange all leaves representing edges directed into vertex k+1 (by reflecting Q-nodes and permuting the children of P-nodes) to appear consecutively in the frontier of the PQ-tree T before the reduction for vertex k+1 begins.

We proceed by induction on k, the st-number of the vertex we are dealing with. For vertex 1, it is clear that T is equivalent to $\hat{B}_1$. Since we can arrange all virtual vertices labelled 2 of the bush form $\hat{B}_1$ to appear consecutively in its frontier, and since we can arrange all leaves

representing edges directed into vertex 2 to appear consecutively in the frontier of the PQ-tree before reduction for vertex 2, the above statement holds for k = 1.

Assume that the above statement holds for vertices with st-number less than or equal to k. Suppose that we are reducing a PQ-tree T with respect to a vertex k+1 ≥ 3, to obtain Reduce(T,S). Then, observe that, in Reduce(T,S) we remove the full children and add a new P-node with children being edges directed out of v_k. Denote this PQ-tree by T'. In the corresponding Bush Form $\hat{B}_k$, we are transforming $\hat{B}_k$ (by reflecting 2-connected components and permuting around cut-vertices) to obtain another Bush Form $\hat{B}_{k_1}$, where all leaves with label k+1 appear consecutively. We then pull down vertex v_k, and add the virtual vertices and virtual edges to the Bush Form, to obtain Bush Form $\hat{B}_{k+1}$. By the inductive hypothesis, we have that the PQ-tree T and Bush Form $\hat{B}_k$ were equivalent. Theorem 2.7 in Section 2.3, together with Theorem 2.8 in this section complete the proof, since then Bush Form $\hat{B}_{k+1}$ and PQ-tree T' must be equivalent.

Now, if the reduction for vertex k+1 fails, then we are unable to gather the virtual vertices labelled k+1 in the corresponding bush form $\hat{B}_k$ together. So, by Corollary 2.3, G is non-planar ❑

Section 2.5

Implementation of PQ-trees in linear time

We may break the algorithm into two mutually dependent passes. The first matches a template with the particular node, possibly using some equivalence transformations, and the second pass applies the replacement pattern. The only restriction in the second pass is that all children of a node must be processed before template matching of the parent node itself can take place; obviously we need to know the status and types of the children nodes before attempting a template match. This restriction can be incorporated into the algorithm by using a queuing mechanism, where children are queued before parents.

The efficiency of this algorithm lies in the fact that we do not consider the entire tree each time we perform a reduction on the tree. In fact, we avoid all processing of empty nodes to obtain an algorithm of linear complexity. The *pruned pertinent subtree* of T with respect to S is the smallest connected subgraph (not necessarily a proper PQ-tree) which has S as frontier; we denote it by Pruned(T,S). Pruned(T,S) of T in Figure 2.14 if S = {a, b, d} is shown in Figure 2.37.

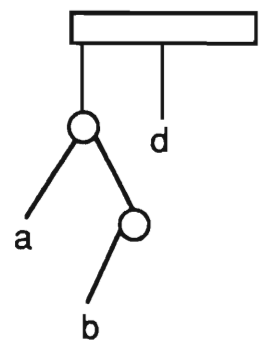


Figure 2.37 : Pruned(T,S) of T from Figure 2.12 with S = {a, b, d}

We note that the root of Pruned(T,S) is Root(T,S). Now, the actual implementation uses a two pass algorithm. The first pass, called the *bubble pass*, identifies nodes of Pruned(T,S) and marks them, whereas the

second pass, called the *reduction pass,* performs the actual reduction process of template matching and replacing to build Reduce(T,S). The entire algorithm is called a *pruned reduction* because it only looks at the pruned pertinent subtree whilst performing the reduction. The name of the first pass is derived from the process of bubbling up of information about Pruned(T,S) to ancestor nodes.

Since we never look at empty nodes, we recognise empty nodes by the absence of labels and their absence from the queue. A *pertinent child* of a node is a child which is either a leaf which belongs to the set S, or a node, some (or all) of whose descendant leaves are in S, i.e. the child is part of Pruned(T,S). During the bubble pass, as well as marking the particular node as being part of Pruned(T, S), we store a count of the number of pertinent children which this node has. This indicates the number of children which must be processed first before we may process this node for template matchings. In the reduction pass, each time we process a node we decrement the count of its parent. If the count reaches zero, then we know that we can process the parent, since all its pertinent children have been matched, and so we add the parent to the queue.

An important consideration in processing the tree is that during the reduction pass, a number of replacement patterns involve assigning new or different parents to nodes. Consider, for example, the Template Q_2 and the case where we have one partial child. Every child of the partial child is assigned a new parent, namely the parent of the partial child. This reassignment of parents occurs frequently during the reduction pass, and could easily involve processing the entire tree. Thus, to keep the complexity of the algorithm low, it is important that we try to avoid this passing as much as possible. With reference to the templates, we can see that every template applied to Pruned(T,S) at a node different from Root(T,S) labels the replacement node either full or partial.

If the node is labelled full, in which case Templates P_1 and Q_1 are applied, it does not involve any work other than a labelling to note that the node is indeed full. If the replacement node is labelled partial, then the

replacement pattern has a new parent being a Q-node, in which case Templates P_3, P_5 and Q_2 apply.

Thus it makes sense to avoid the assignment of parent pointers to the children in Templates P_3, P_5 and Q_2. To this end we adopt the following scheme. The children of a P-node always know who their parents are. For Q-nodes, the only children which permanently know who their parents are, are the endmost children of that node. The rest of the children are only assigned parent pointers on a 'need-to-know' basis. During the bubble pass, we assign valid parent pointers only to the pertinent children. This method ensures that we avoid the processing of all children of a node to assign them to a new Q-node parent.

We now describe the modifications for the bubble pass to assign parent pointers to the pertinent children of Q-nodes. Initially, all nodes are marked *blank*. Note that this step is implicit - we don't actually do it. Once a node has been placed onto the queue, it is marked *queued*. When we remove it from the queue, we determine whether it can receive a valid parent pointer by looking if one of its immediate siblings has a valid parent pointer (or, of course, if the node itself is endmost or if it has a P-node as a parent). If it can receive a valid parent pointer, then we mark the node as *unblocked*. If the parent of the node cannot be determined, then we note that it is *blocked*. Further, we store a total number of nodes which are blocked, *blocked_nodes*, and a count of the total number of blocks of blocked nodes, where a block is a consecutive group of blocked nodes. This count is called the *block count*, and is initially zero (no blocked nodes). A node may be classified unblocked if one of the following three situations apply.

(i) It has no immediate siblings. This implies the parent is a P-node, and so, by definition, the node has a valid parent pointer.
(ii) If the node has an immediate sibling which is unblocked. In this case, we can obtain the correct parent pointer from the sibling.
(iii) The node only has a single immediate sibling. This implies that the node is an endmost child and consequently always has a valid parent pointer.

In the discussion that follows we shall discuss the various cases when a node is blocked and unblocked. In the diagrams we shall use cross-hatching to indicate the node currently being processed, whereas shaded nodes are nodes which have been processed and unshaded nodes are nodes which are not yet processed. If a child has a valid parent pointer, then we shall indicate it in the diagram by means of a line to the parent.

If the node is unblocked, then we need to look at the immediate siblings. If any immediate sibling is blocked, then we need to decrement the block count. We then move through the block of blocked siblings while assigning valid parent pointers to the blocked nodes and unblocking them. The value of blocked_nodes decreases by the size of the block whose nodes we are unblocking. In Figure 2.38, we show the above case where we unblock a block of blocked nodes.

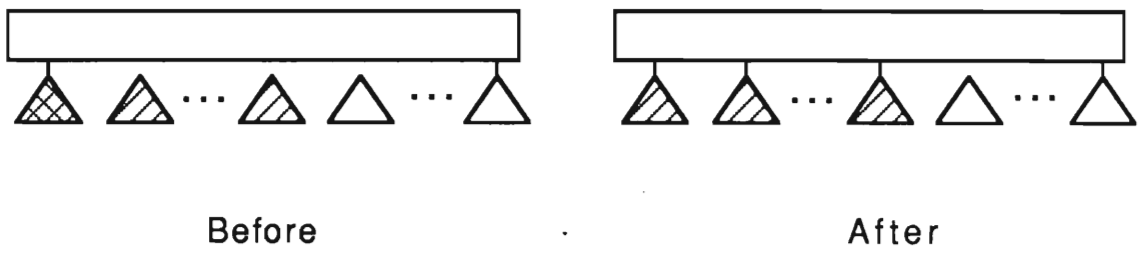


Figure 2.38 : Before and After unblocking to decrement Block Count

If the node cannot be unblocked, the maintenance of block count requires some thought. There are three cases, which are determined by the numbers (0, 1 or 2) of adjacent blocks of blocked nodes. This number is easily determined by looking at the immediate siblings, and counting the number of blocked siblings. The first case is when the node is not adjacent to any blocks of blocked nodes. In this case we increment the number of block count by one. Figure 2.39 shows this case.

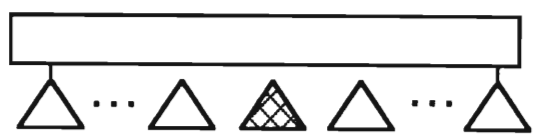


Figure 2.39 : Block count increases by one

If the node is adjacent to a block, then we can append the blocked node onto that block and so the block count remains unchanged, see Figure 2.40.

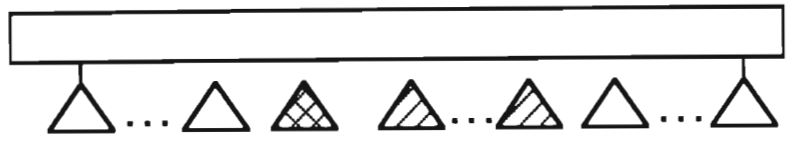


Figure 2.40 : Block count remains unchanged

Lastly, if both immediate siblings are blocked, then we can decrease the block count by one. This is because the new blocked node together with its two adjacent blocks form a new block. This final case is shown in Figure 2.41.

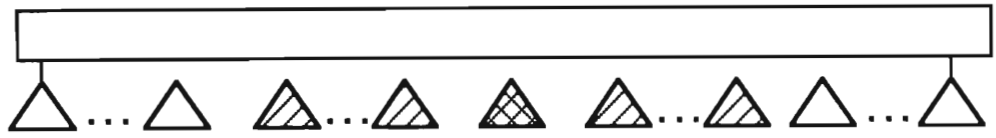


Figure 2.41 : Block count decreases by one

This process of determining the blocks of blocked nodes, and assigning valid parent pointers also provides a preliminary check on the viability of the reduction pass. Consider what happens when we end the bubble pass with more than one block of blocked nodes. In this case, the reduction would end unsuccessfully since there are two groups of pertinent nodes which are in the middle of a sequence of Q-node children, and are bounded on either side by empty siblings. This case will not fit any of the templates. However, the case of one block of blocked pertinent children may still be valid. For successful matching in this case, their parent needs to be Root(T,S) and Template Q_3 will match successfully. To process this case the children need to have valid parent pointers, but we want to avoid processing the empty nodes to obtain the valid parent pointers. Therefore, we create a dummy parent which we call a *pseudo node*, and let

it be the parent of the blocked children. The pseudo node is removed after the reduction pass is completed. We shall return to discuss the pseudo node after the bubble pass procedure has been detailed.

Note that we must reset the marks of all nodes affected by the labelling process performed during the bubble pass before we can start a new pass of the reduction algorithm. Again, we shall return to this problem of reinitialisation of certain fields later.

Ideally, we would require the bubbling pass to halt at Root(T,S), the root of the pertinent subtree. We want to keep the number of extraneous nodes which are processed to a minimum. Unfortunately, the position of Root(T, S) is difficult to detect, since we could still be processing nodes further down Pruned(T,S). Thus, we must continue processing until there is only one node in the queue. This approach will work until processing reaches the top of the tree. If processing reaches the top of the tree and we still need to perform processing further down the tree, then problems will occur, since we have only one node, but it is not Root(T, S). Thus, the parents of the node must be processed as well. To aid in the detection of proper end conditions for the bubble pass, we keep a flag called *Off_the_Top* to denote that we have reached the top of the tree, but that we must still continue processing until the proper conditions are met. Off_the_Top takes on the values one or zero, depending on whether we have reached the top of the tree or not, respectively. In effect, Off_the_Top simulates the presence of ancestors of the root of T.

There are two cases where Off_the_Top is not set and we can end processing. In the first case, if the queue has one node to process and the block count is zero, then that node must be Root(T, S) or an ancestor of Root(T, S). In the second case, if the block count is one and the queue is empty, then we must end processing, since the parent of the blocked nodes is Root(T, S), and will match Template Q_3.

If we reach the top of the tree with a block count of one and the queue is empty, then the tree is irreducible because there is a group of nodes blocked lower down in the tree. Hence, the block is not at Root(T,S) which

is the only valid node to have such a block of blocked nodes as children (recall that Template Q_3 is only valid for Root(T,S)). As Booth and Lueker [BL76] note, if the tree is irreducible then this bubble procedure may perform a great deal of extra work. However, in the worst case the procedure can process the entire tree, and since the reduction pass will halt at this point in any case, one more pass over the tree does not affect the overall algorithm complexity.

Before giving the actual algorithm for the bubble pass, we will discuss the data structures and global variables involved in the algorithm. First, we give the global variables used during either or both passes of the algorithm:

Block_Count. A count of the number of groups of blocked nodes. It is used during the bubble pass. Initially it is set to zero.

Blocked_Nodes. The total number of blocked nodes. It is used during the bubble pass. Initially it is set to zero.

Off_the_Top. A flag which indicates that processing has passed the top of the tree, but that we are still busy processing the tree. It is used during the bubble pass. Initially it is set to zero.

Queue. A first-in first-out data structure which is used during both the bubble pass and the reduction pass to store the nodes which need processing. Queue is initially empty.

We now describe the PQ-tree data structure which we use to represent a particular node of a PQ-tree. The data structure is essential for implementing the algorithm in linear time. We list below each field which a particular node of a PQ-tree has. Certain fields are used only if the node in question is of a certain type, and where appropriate we note this.

Child_Count. This field is only used for P-nodes, and refers to the total number of children which the node has.

Circular_Link. A doubly linked list storing the pointers to the children of the node. The order is arbitrary. The field is used only for P-nodes.

Endmost_Children. We store pointers to the two children of a Q-node which each have only one immediate sibling. Obviously, we only use this field for Q-nodes.

Full_Children. For either P-nodes or Q-nodes, we store a list of pointers to the full children of this node. This list is initially empty, and is adjusted during the reduction pass as the full children are identified.

Full_Children_Count. Again for either P-nodes or Q-nodes, this field stores the size of the Full_Children list. The field is initially set to zero.

Immediate_Siblings. This unordered list contains zero elements for children of P-nodes, one element for endmost children of Q-nodes and two elements for interior children of Q-nodes. Each element is a pointer to the relevant sibling which is an immediate sibling of the node.

Label. This value notes the state of the node (i.e. empty, full or partial). Initially, it is set to empty.

Mark. Used for the bubble pass, values range through none, queued, blocked or unblocked. Initially, it is set to none.

Parent. When valid, this field is a parent pointer to the parent of the node.

Partial_Children. This list stores from zero (no partial children) to two elements, where each element points to the relevant child labelled partial. There cannot be three partial children, since in this case the tree would be irreducible and processing would stop. Initially, of course, this field is empty (no partial children).

Pertinent_Child_Count. Stores the number of pertinent children which this node has. Initially zero, it is set during the bubble pass and used in the reduction pass to recognise when we may process that node.

Pertinent_Leaf_Count. Initially zero, it is set in the reduction pass. This variable indicates the total number of leaves which are pertinent in the subtree of the pertinent subtree which has the node currently being processed as root. We use this field to recognise Root(T,S) in the reduction pass,

since Root(T,S) would be the first node reached during processing with Pertinent_Leaf_Count equal to the size of the set S (the pertinent leaf set).

Type. A field identifier which describes whether the node is a leaf, a P-node or a Q-node.

Circ_List_Posn. Only valid when the node has a P-node parent. This variable points to the element in the Circular_Link list (see above) of the parent which points to this node. Booth and Lueker [BL76] fail to adequately justify the linearity of parts of the program. We need this field to extract the full children from the Circular_Link list of the parent efficiently. We shall justify the use of this field more thoroughly later.

Now we can present the bubble procedure. Note that we do not consider all the details. In particular we do not consider details about the processing of the pseudo node. We defer these points until after the algorithm.

Algorithm 2.12: Reduction_Algorithm_Bubble_Pass (T, S)

{ Given a PQ-tree T, and a subset S of the leaves of T, find
Pruned(T, S) and initialise fields, as discussed above }

```
Queue = empty
Block_Count = 0
Blocked_Nodes = 0
Off_the_Top = 0
for each Leaf ∈ S,
   place Leaf onto Queue
while |Queue| + Block_Count + Off_the_Top > 1 do
      if |Queue| = 0          { we are at top of tree and block count > 0 }
         then
            T = ø             { return the null tree because tree irreducible }
            Halt              { and stop the algorithm }
      Remove Current from head of Queue
      Mark(Current) = Blocked      { assume blocked until we unblock }

{ Get the total number of blocked and unblocked siblings of Current }

      Blocked_Sibs =  {All Immediate Siblings of Current
                                          with Mark = Blocked}
      UnBlocked_Sibs = {All Immediate Siblings of Current
                                            with Mark = Unblocked}
```

```
{ Now check if we may unblock Current }

    if |UnBlocked_Sibs| > 0
       then        { we may unblock Current }
          choose any Sibling ∈ UnBlocked_Sibs
          Parent(Current) = Parent(Sibling)    { tell it the parent }
          Mark(Current) = Unblocked            { note unblocked }
       else     { See if Current has P-node parent or is endmost }
          if |Immediate_Siblings(Current)| < 2
             then Mark(Current) = Unblocked

{ If Current is unblocked, then unblock siblings and queue Parent }

    If Mark(Current) = Unblocked
       then                  { check if we can unblock siblings }
          Current_Parent = Parent(Current)
          if |Blocked_Sibs| > 0
             then  { only one adjacent block of blocked siblings }
                Blocked_List = Max consecutive set of
                               blocked siblings
                for every Element of Blocked_List do        { unblock }
                      Mark(Element) = UnBlocked
                      Parent(Element) = Current_Parent
                      Pertinent_Child_Count (Current_Parent) =
                      Pertinent_Child_Count (Current_Parent)+1
          If Current_Parent = Empty
             then Off_the_Top = 1      { overflow off top of tree }

             else      { tell parent that Current is a pertinent child }
                Pertinent_Child_Count (Current_Parent) =
                   Pertinent_Child_Count (Current_Parent) + 1
                if Mark(Current_Parent) = None
                   then
    { queue the parent even though pertinent children
          may change later (other children may be pertinent) }
                      Add_to_Queue (Current_Parent)
                      Mark(Current_Parent) = Queued
                { adjust  number of Block_Count and Blocked_Nodes}
             Block_Count = Block_Count - |Blocked_Sibs|
             Blocked_Nodes = Blocked_Nodes -  |Blocked_List|
       else

{ the node is blocked so adjust Block_Count and Blocked Nodes }

             Block_Count = Block_Count + 1 - |Blocked_Sibs|
             Blocked_Nodes = Blocked_Nodes + 1
```

end

Note that the list processing to unblock an adjacent blocked sibling may be done in time proportional to |Blocked_List|. This is easy to see, since we can traverse the block of blocked siblings by using the immediate sibling

fields. We keep adding siblings to Blocked_List until we encounter a sibling which is not blocked. Further, note that at most one immediate sibling of Current is blocked if Current is unblocked (since Current must have been endmost or adjacent to an unblocked node to become unblocked). Thus it suffices to generate one Blocked_List of blocked siblings.

We shall now discuss the pseudo node case. If the procedure ends with a block count of greater than one, then, as explained earlier, the tree is irreducible for S. If we get a block count of one and only one blocked node, then we do not have to be concerned with invalid parent pointers, since that node will be Root(T,S) (i.e. we will never have to assign parent values for Root(T, S)). However, if we get a block count of one and at least two blocked nodes, then those nodes still blocked must receive proper parent pointers and so we must create a pseudo node parent. The introduction of such a node is necessary since we need to have a Root(T,S), and since all processing of empty children should be avoided. Instead of processing the empty nodes, we merely give the pertinent children a temporary parent, which we destroy at the end of the reduction. To implement this, we modify the bubble procedure to keep a list, called Blocked_Nodes_List which stores all the blocked nodes in the tree. Then, at the end of the procedure, if we must create a pseudo node, called Pseudo_Node say, then we add all the pertinent children in the Blocked_Nodes_List as children to Pseudo_Node. The endmost children of Pseudo_Node are the two nodes from Blocked_Nodes_List which have only one immediate sibling which is blocked (the other sibling will be blank). Consider now what happens in the normal algorithm, where every node is always assigned a valid parent pointer. In this case, Root(T,S) would only match Template Q_3, since no pertinent children of the parent Q-node are endmost. For this reason we match the pseudo node only to Template Q_3, even though it could possibly match another template (possibly Templates Q_1 or Q_2). This ensures that the sibling chains between the pertinent children and the empty children (which are implicitly present in the pseudo node's case) are properly maintained.

It is easy to derive an expression for the overall complexity of the bubble pass. We say that a PQ-tree T has order p, written |T|, if there are p nodes in T. Here we include both internal nodes and leaves.

Theorem 2.10: The bubble pass (Algorithm 2.12) of the reduction algorithm requires O(|Pruned(T,S)|) steps.
Proof: First, we note that the main loop of the algorithm (the while statement) is performed at least |Pruned(T,S)| times, once for each node of Pruned(T, S). Also, the while loop may be performed for nodes which are ancestors of Root(T,S). This happens for at most the distance (in terms of ancestors before Root(T, S)) between the furthest leaf and Root(T,S), which in any case is not more than |Pruned(T,S)| times extra. Thus the loop is iterated no more than O(|Pruned(T,S)|) times. Secondly, since every pertinent node is blocked at most once, the for statement which unblocks the node is certainly never executed more than a total of O(|Pruned(T,S)|) times throughout the entire algorithm. Overall complexity of the algorithm is thus O(|Pruned(T,S)|). ❑

We can now present the reduction pass procedure. From the bubble pass each pertinent node knows its parent, and in addition we can determine the Root(T,S). We do not concern ourselves with the Templates P_0 and Q_0 explicitly, since we are only processing the pertinent subtree. Thus the template matching for the rest of the tree which is not part of the pertinent subtree is implicit.

Algorithm 2.13: Reduction_Algorithm_Reduction_Pass (T, S)

```
{ Given a PQ-tree T, preprocessed by the bubble pass
  (Algorithm 2.12), if possible, this algorithm reduces T so
  that for all T' ≡ T, elements of S appear consec. in
  Frontier (T') }

Queue = empty
for every Leaf ∈ S do
   Add Leaf to Queue
   Pertinent_Leaf_Count (Leaf) = 1

while |Queue| > 0 do
      Remove Current from head of Queue
```

```
        if Pertinent_Leaf_Count(Current) < |S|
           then
    { we are not at Root(T,S) yet }
              Current_Parent = Parent(Current)

    { Tell Parent about number of pertinent leaf descendants }
              Pertinent_Leaf_Count(Current_Parent) =
                 Pertinent_Leaf_Count(Current_Parent) +
                    Pertinent_Leaf_Count(Current)

    { Check if all pertinent children of Parent have been matched }
              Pertinent_Child_Count(Current_Parent) =
                 Pertinent_Child_Count(Current_Parent) - 1
                          { if pertinent_child_count = 0 then all
                             pertinent children have been queued }
              if Pertinent_Child_Count(Current_Parent) = 0
                 then Add_to_Queue (Current_Parent)

    { Attempt to match template to Current }
              if not Template_L1(Current)
                 then if not Template_P1(Current)
                 then if not Template_P3(Current)
                 then if not Template_P5(Current)
                 then if not Template_Q1(Current)
                 then if not Template_Q2(Current)
                 then        { no templates match - tree is irreducible }
                    T = ø          { return the null tree }
                    Halt           { and stop the algorithm }
           else
    { we are at Root(T,S), attempt to match Root to a root template }
              if not Template_L1(Current)
                 then if not Template_P1(Current)
                 then if not Template_P2(Current)
                 then if not Template_P4(Current)
                 then if not Template_P6(Current)
                 then if not Template_Q1(Current)
                 then if not Template_Q2(Current)
                 then if not Template_Q3(Current)
                    then     { no templates match - tree is irreducible }
                       T = ø          { return the null tree }
                       Halt
end
```

The specific order in which the algorithm attempts to match templates is important. We use information about the failure of previous template matching to expedite the testing for the next template matching. We show that the complexity of Algorithm 2.13 is O(|Pruned(T,S)|). This bound we achieve by avoiding any processing of nodes outside Pruned(T,S) (i.e. we avoid processing any empty nodes). Consider for example Template P_3

(see Figure 2.20), where we change the current node to a Q-node. It seems as though we have to assign each empty node a new parent. This is not actually done. Instead, we remove references to the full nodes from the current node's circular list, and make the old node the 'new' parent of the empty nodes in the diagram . This way we avoid having to assign a new parent to each empty node. That is, in effect we avoid any processing of the empty nodes. We shall consider each template in turn, and where necessary we shall elaborate on the complexity of the steps involved. The first template is Template L_1, a pertinent leaf. No matching is required, apart from checking if the node is indeed a leaf. Since we only queue nodes of Pertinent(T,S) we know that all leaf nodes on the queue are pertinent.

Algorithm 2.14: Function Template_L1 (Current) : Boolean
{ Check if Template L_1 matches the node Current }

```
if Type(Current) ≠ Leaf
   then Template_L1 = false
   else
      Data_Label(Current) = full
      if Current ≠ Root(T,S)              { Update full list of parent }
         then
                                          { Tell Parent about full child }
            Increase Parents count of full children by one
            Add Current to list of full children of Current's parent
      Template_L1 = true
```

end

Algorithm 2.14 does not need much discussion, except to note that the addition into the parent's full children list may be done by a simple addition to the head of that linked list.

Template P_1 is for P-nodes which have all their children full. We merely compare the Full_Kids_Count against Child_Count for the matching phase. The replacement pattern is merely a labelling.

Algorithm 2.15: Function Template_P1 (Current) : Boolean
{ Check if Template P_1 matches the node Current }

```
if Type(Current) ≠ P_Node
   then Template_P1 = false
   else if Full_Kids_Count(Current) ≠
                Child_Count(Current)
        then Template_P1 = false
        else                                  { we have matched Current to P1 }
           Data_Label(Current) = full
           if Current ≠ Root(T,S)             { update parent full children }
              then
                                              { Tell Parent about full child }
                 Increase Parents count of full children by one
                 Add Current to Parents list of full children
           Template_P1 = true
```

end

Again, this algorithm is straightforward. The next template, Template P_2 is specifically for Root(T,S). First we check that the P-node has no partial children and then we use the fact that Template P_1 has failed to deduce that the template is matched. Replacement requires gathering all full nodes under a single node.

Algorithm 2.16: Function Template_P2 (Current) : Boolean
{ Check if Template P_2 matches the node Current }

```
if (Type(Current) ≠ P_Node) or
      (|Partial_Kids(Current)| ≠ 0)
   then Template_P2 = false
   else                                              { Matched! }
      { Now we check for only one full child }
      if Full_Kids_Count(Current) = 1
         then Root(T,S) = Full_Kid(Current)  { New root }

         else
{ Move all full children from Current to children of Full Node}
            Strip_Full_Nodes (Current, Full_Node)
{  Add Full_Node to list of Current's children }
            Add_to_Circle_Link (Current, Full_Node)
{ Note Full_Node now an extra child of Current and loss of others }
            Child_Count(Current) =
               Child_Count(Current) - Full_Kids_Count + 1
{ Adjust Parent Pointer of Full Node to reflect parent }
               Parent(Full_Node) = Current
      Template_P2 = true
```

end

The procedure Strip_Full_Nodes removes all full children from the circular list of children of Current. The children are then given a new parent, called Full_Node. If there was only one full child, then Full_Node is that node. Through the use of the Circ_List_Posn field and the Full_Kids list we can perform the removal of the children from the circular list in O(|Pertinent_Children of Current|) operations. The addition of Full_Node to the circular link of Current is merely a simple addition into the doubly linked list. Without having the Circ_List_Posn field, the operation count of removing the full children from the circular list of Current increases to O(|Children of Current|), since we would have to possibly scan the entire doubly linked list to remove the full children. It is difficult to see how else we could efficiently remove the full children from the circular list without scanning through some or all of the empty children. Thus Circ_List_Posn is an essential field for the complexity of the algorithm to be linear. Notice also how we avoid processing any of the empty children, and that we change Root(T,S) to the new parent of the full children.

Template P_3 deals with the same matching, but is for nodes which are not Root(T,S). The replacement pattern is more difficult, and consists of "changing" Current into a Q-node with two children, one of which is full and the other is empty.

Algorithm 2.17: Function Template_P3 (Current) : Boolean
{ Check if Template P_3 matches the node Current }

```
if (Type(Current) ≠ P_Node) or
      (|Partial_Kids(Current)| ≠ 0)
   then Template_P3 = false
   else                                                { Matched }
      Make a new node called New_Root
{ The new Q-node will replace Current }
      Type(New_Root) = Q_Node
{ Note Number_Empty to keep correct tallys later }
      Number_Empty = Child_Count(Current) -
                        Full_Kids_Count(Current)

{ See below for a full discussion }
      Replace_Node_Partial (Current, New_Root)
         { replace every instance of Current in immediate sibling
           fields and Current's parent to New_Root }
```

```
                Strip_Full_Nodes (Current, Full_Node)
                        { Move full children from Current to children
                          of node called Full_Node }

            { Note Full_Node is a full child of New_Root }
                Add_to_Full_List (New_Root, Full_Node)
                Parent(Full_Node) = New_Root{ Add Full_Node on as child }

            { Now set up the empty child of New_Root }
                if Number_Empty = 1
                   then Empty_Node = Only_child_left (Current)
                   else
                      Empty_Node = Current
                      Child_Count(Empty_Node) = Number_Empty
                      Data_Label(Empty_Node) = Empty
                Parent(Empty_Node) = New_Root { tack empty node as child }
                Add_Sibling (Empty_Node, Full_Node) { they are siblings }
                Data_Label(Full_Node) = Full
                Rightmost_Kid(New_Root) = Full_Node { they are endmost }
                LeftMost_Kid(New_Root) = Empty_Node { in any order! }
                Template_P3 = true
end
```

The procedure call Replace_Node_Partial (Current, New_Root) will find every instance of Current in the Immediate_Sibling chains of Current's immediate siblings, if Parent(Current) is a Q-node, and every instance of Current in the Circular_Link field of Parent(Current) if Parent(Current) is a P-node, and replace them by the new partial node New_Root. Again, this is easy by, in the first instance, merely checking the Immediate_Siblings list of each sibling of Current, and in the second case through the use of the Circ_List_Posn field. Add_Sibling accepts two Nodes and adds them to each other's Immediate_Sibling lists. Only_Child_Left(Current) returns the solitary remaining child in the circular list of Current. Notice the small amount of extra work (via Empty_Node) to ensure that chains of nodes which only have a single child do not occur. This is done, firstly, for efficiency. Secondly, it is done to keep the PQ-tree proper, in that after reduction all non-pertinent nodes satisfy the requirements of nodes for proper PQ-trees.

Template P_4 is specifically for Root(T,S). The matching phase has to check for exactly one partial child. In the replacement this partial child becomes the new Root(T,S). Any full children are gathered at the full side of the partial child. The empty children are left as children of Current.

Algorithm 2.18: Function Template_P4 (Current) : Boolean
{ Check if Template P_4 matches the node Current }

```
if Type(Current) ≠ P_Node
   then Template_P4 = false
   else if |Partial_Kids(Current)| ≠ 1 { must be exactly one }
         then Template_P4 = false
         else
            Partial_Kid = The only partial child
            if Full_Kids_Count(Current) > 0
               then Strip_Full_Nodes (Current, Full_Node)
            { Now, add Full_Node to the correct end of Partial_Child }
            if Data_Label(Rightmost(Partial_Child)) = full
               then
                  Add_Sibling (Rightmost(Partial_Child),
                                    Full_Node)
                  Rightmost_Child(Partial_Child) = Full_Node
               else
                  Add_Sibling (Leftmost(Partial_Child),
                                    Full_Node)
                  Leftmost_Child(Partial_Child) = Full_Node
            Root(T,S) = Partial_Child
            Template_P4 = true
```

end

Once again, we may efficiently remove the full children of Current and assign them to Full_Node through the use of Circ_List_Posn fields. Note also that we have to reassign Root(T,S) to the partial child.

Template P_5 matches at a node which is not Root(T,S) and has exactly one partial child. This partial child becomes the new Current node.

Algorithm 2.19: Function Template_P5 (Current) : Boolean
{ Check if Template P_5 matches the node Current }

```
if Type(Current) ≠ P_Node
   then Template_P5 = false
   else if |Partial_Kids(Current)| ≠ 1 { exactly one  partial child}
      then Template_P5 = false
      else                                          { matched }
            { We replace Current by the partial child}
         New_Root = Partial_Kids(Current)

      { Number_empty is a temporary count of Current's empty children}
            Number_Empty = Child_Count(Current) -
```

```
                                    Full_Kids_Count(Current) -
                                    |Partial_Kids(Current)|

                { Obtain pointers to the endmost children of New_Root }
                Let full endmost child be Full_Kid

                   { Remove New_root from Currents Circular_List }
                Delete_from_Circular_List (Current, New_Root)

                Replace_Node_Partial (Current, New_Root)
                if Full_Kids_Count(Current) > 0
                   then          { add full_nodes onto end of New_Root }
                      Strip_Full_Nodes (Current, Full_Node)
                      Add_Sibling (Full_Kid, Full_Node)

            { Adjust left or right endmost child of New_Root to Full_Node }
                      Adjust relevant Endmost of New_Root to Full_Node

                if Number_Empty > 0
                   then       { add empty nodes onto end of New_Root }
                      if Number_Empty = 1
                         then Empty_Node=
                                 Only_child_left(Current)
                         else
                            Empty_Node = Current
                            Data_Label(Empty_Node) = Empty
                            Child_Count(Empty_Node) = Number_Empty
                      Parent(Empty_Node) = New_Root
                      Add_Sibling (Empty_EndMost_Child, Empty_Node)
                      Adjust relevant Endmost of New_Root to Empty_Node
                Template_P5 = true
```

end

The code for matching a pattern with Template P_5 and making the relevant replacement has complexity O(|Pertinent Children of Current|). Once again, we can see that, through the use of the Circ_List_Posn field, Delete_from_Circular_List may be performed efficiently. Again, all other procedure calls are straightforward and have complexity O(|Pertinent Children of Current|).

Template P_6 is the last template for P-nodes, and applies only to Root(T,S). Here we have exactly two partial children. They are merged into one partial child with any full children of the current node fixed in between the two groups of full nodes of the partial children.

Algorithm 2.20: Function Template_P6 (Current) : Boolean

```
{ Check if Template P6 matches the node Current }

if Type(Current) ≠ P_Node
   then Template_P6 = false
   else if  |Partial_Kids(Current)| ≠ 2
      then Template_P6 = false
      else                                          { matched }
         Let Partial1 and Partial2 be the two partial children
         Number_Empty = Child_Count - Full_Kids_Count - 2
         Let Full and Empty endmost children of Partial1 be
               Empty_P1, Full_P1
         and of Partial2 be
               Empty_P2, Full_P2

{ we now adjust Partial1 so that it becomes Root(T, S) }
         if Full_Kids_Count > 0
            then
{ If Full children then insert between 2 groups of full sibs from partials }
               Strip_Full_Nodes (Current, Full_Node)
               Add_Sibling (Full_Node, Full_P1)
               Add_Sibling (Full_Node, Full_P2)
            else
{ No full children so just join two groups of full children from partial
children }
               Add_Sibling (Full_P1, Full_P2)

{ Empty_P2 is new endmost child of the new joint partial node }
{ Empty_P1 is still an endmost child of the new joint partial node }
         Parent(Empty_P2) = Partial1
         Adjust relevant Endmost child of Partial1 to Empty_P2

{ Partial2 is now deleted }
         Delete_from_Circular_List (Current, Partial2)
         Root(T,S) = Partial1

{ we now ensure that the tree remains a proper PQ-tree }
         if Number_Empty = 0 { check for chains }
            then        { Current just had two partial nodes }
               Replace_Node_Partial (Current, Root_Node)
         Template_P6 = true
```

end

It is easy to verify that the above procedure can be performed in O(|Pertinent Children of Current|) operations.

The remaining three templates are Templates Q_1, Q_2 and Q_3. Again, the code is quite straightforward, however the checking for correct matching is sometimes long and tedious. Template Q_1 is analogous template to

Template P_1, i.e. Current's children are all full. The problem of matching though is not as simple as merely comparing the count of full children against the empty children count, since such a count is not kept for Q-nodes. However, the next function indicates how the matching and replacement can be performed using O(|Pertinent Children of Current|) operations.

Algorithm 2.21: Function Template_Q1 (Current) : Boolean
{ Check if Template Q_1 matches the node Current }

```
if (Type(Current) ≠ Q_Node) or
      (Current = Pseudo_Node)
   then Template_Q1 = false
   else
      if (Data_Label(Rightmost_Kid(Current)) ≠ Full) or
         (Data_Label(Leftmost_Kid(Current)) ≠ Full)
         then Template_Q1 = false
         else
            Count number of full children with less than 2 full sibs
            if (Count ≠ 2) or
                  (Data_Label(Immediate_Siblings
                     (RightMost_Kid(Current))) ≠ Full)
               then Template_Q1 = false
               else                        { Successfully Matched! }
                  Data_Label = full
                  if Current ≠ Root(T,S)
                     then          { Tell Parent about full child }
                        Increase count of full children of
                           Parent(Current) by one
                        Add Current to Parents list of full children
                  Template_Q1 = true
```

end

The code for Template Q_1 is straightforward, involving only a traversal of the full children list of Current to count the total number of children with less than two siblings. This procedure has complexity O(|Pertinent children of Current|) (by traversal of full children list). If Current is not full, then there will be more than two "endmost" (with respect to other full children) full children, or there could be exactly two full children which are both endmost with only empty nodes as interior nodes. Lastly, recall, from Template P_1, that adding a full child to the list of Full_Kids of the parent is a simple addition to the head of the linked list.

Template Q_2 is for a Q-node with at most one partial child. This partial child becomes merged into the parent's children.

Algorithm 2.22: Function Template_Q2 (Current) : Boolean
{ Check if Template Q_2 matches the node Current }

```
if (Type(Current) ≠ Q_Node) or
        (Current = Pseudo_Node)
   then Template_Q2 = false
   else if |Partial_Kids(Current)| > 1
        then Template_Q2 = false
        else
           if |Partial_Kids| ≠ 0
              then Partial_Child = Partial_Kids(Current)
              else Partial_Child = Empty
           if Full_Kids_Count(Current) > 0
              then
                 { check the full children are continuous and at one end }
                 if a full child is not endmost
                    then
                       Template_Q2 = false  { no full children endmost }
                       Exit                 { try another template }
                 if full children are not all consecutive after endmost
                    then
                       Template_Q2 = false  { full children not consec. }
                       Exit                 { try another template }

                 if Partial_Child ≠ Empty
                    then      { check it is neighboring full children }
                       if a sibling of Partial_Child not full
                          then
                             Template_Q2 = false
                             Exit     { partial child not following full}

                 else
{ There are no full children - check partial child is endmost }
                    if Partial_Child not endmost
                       then
                          Template_Q2 = false
                          Exit          { match failure - try next  template }

{ We have succesfully matched }
              Data_Label = Partial
              Add_to_Partial_list (Parent(Current), Current)
              if Partial_Child ≠ Empty
                 then { merge with siblings }
                   For Partial_Child, let the siblings be
                         Full_Sib, Empty_Sib
                   and the endmost children be
                         Full_Kid, Empty_Kid
```

```
{ Add sibling link to full sibling }

                    if Full_Sib ≠ Nil
                       then
{ Replace the references in Full_Sib  to Partial_Child with
  Full_Kid and add reference in Full_Kid to Full_Sib }
                          Add_Replace_Siblings (Full_Kid,
                                          Partial_Child, Full_Sib)
                    else
                       Adjust relevant endmost child of Current
                                                   to Full_Kid
{ Either add sibling link to empty sibling or adjust endmost values }
                    if Empty_Sib ≠ Nil
{ Replace reference in Empty_Sib to Partial_Child with
  Empty_Kid and add reference in Empty_Kid to Empty_Sib }
                    then Add_Replace_Sibling (Empty_Kid,
                                   Partial_Child, Empty_Sib)
                    else Adjust relevant endmost child of Current
                                                   to Empty_Kid

{ Last, add Full_Kids(Current) to Partial_Child, remove from Current }
                    Update_Full_Kids (Current,  Partial_Child)
                 Template_Q2 = true
```

end

Notice that for the case where there are no partial child present, the replacement template is merely an update of the parent of Current to inform it that Current is a partial child. The length of the algorithm for Template Q_2 stems mainly from the variety of cases: there can be no empty children or no full children, or both can be present. In any combination with these cases, there may or may not be a partial child. The template is valid in all cases. Add_Replace_Siblings (A, B, C) merely adds C to A's sibling list and replaces B with A in C's sibling list. It is easy to see that the order of the algorithm is O(|Pertinent Children of Current|) by observing that all operations are with full or partial children.

The last template which we need to consider is Template Q_3, when there can be up to two partial children, and the full children and partial children do not have to be endmost. The only restriction we place is that the full children must be consecutive and any partial children must directly precede or follow the full children. Of course, the current node

must be Root(T, S). The techniques employed are exactly the same as for Template Q_2.

Algorithm 2.23: Function Template_Q3 (Current) : Boolean
{ Check if Template Q_3 matches the node Current }

```
if Type(Current) ≠ Q_Node
   then Template_Q3 = false
   else if  |Partial_Kids (Current)| > 2
      then Template_Q3 = false
      else
{ First gather some information }
         if |Partial_Kids| ≠ 0
            then                    { Get the partial children if any }
               Let Partial1 be the first partial child
               If  |Partial_Kids (Current)| = 2
                  then
                     let Partial2 be the second partial child
                  else
                     Partial2 = Empty
            else
               Partial1 = Empty

{ Now perform the matching process }

         if (Full_Kids_Count(Current) > 0)
            then
{ Check that they are consecutive }
               { Note that we cannot have exactly one pertinent child,
                 otherwise it would be Root(T, S) }
               Count number of full children with less than 2 full sibs
               if (Count ≠ 2) or
                  (First full child not adjacent to another)
                     then               { They are not consecutive }
                        Template_Q3 = false
                        Halt            { tree is therefore irreducible }
               Count the Full Neighbors of the Partial Nodes
               if Count ≠ |Partial_Kids(Current) |
                  then  { Partial children are adjacent to full children }
                     Template_Q3 = false
                     Halt               { tree is irreducible }
            else
{ if there are 2 partial nodes they must be adjacent }
               if |Partial_Kids (Current)| = 2
                  then if Partial2 ≠
                              Immediate_Siblings(Partial1)
                           then
                              Template_Q3 = false
                              Halt      { tree is irreducible }

{ Matching now complete }
```

```
            if |Partial_Kids| ≠ 0
               then begin {work to do only if there are partial children }
                  Data_Label = Partial
          { don't have to add to parent's partial list since we are Root(T,S) }
                  Suppose Siblings of Partial1 are
                     Full_Sib, Empty_Sib
                  and the children are
                     Full_Child, Empty_Child

        { Either add sibling link to empty sibling or adjust endmost values }
                  If Empty_Sib ≠ Empty
                     then
                        Add_Replace_Sibling (Empty_Child,
                                 Partial1, Empty_Sib)
                     else
                        Adjust relevant endmost child of Partial1
                                                to Empty_Child

        { We know the full sibling is not endmost, so just adjust siblings}
                  Add_Replace_Sibling (Full_Child,
                                 Partial1, Full_Sib)

        { Adjust list of full children to include Partial1's full children }
               Adjust_Full_Kids (Current, Partial1)
                  if Partial2 ≠ Empty
                     then
                        Suppose Siblings of Partial2 are
                           Full_Sib, Empty_Sib
                        and the endmost children are
                           Full_Child, Empty_Child

        { Either add sibling link to empty sibling or adjust endmost values }
                        if Empty_Sib ≠ Empty
                           then Add_Replace_Sibling (Empty_Child,
                                    Partial2, Empty_Sib)
                           else
                              Adjust relevant endmost child of Partial1
                                                      to Empty_Child

        { We know Full_Sib is not endmost, so just adjust siblings}
                        Add_Replace_Siblings (Full_Child,
                                          Partial2, Full_Sib)

        { Adjust list of full children to include Partial2's full children }
                        Adjust_Full_Kids (Current, Partial2)
                  Template_Q3 = true
end
```

With the use of sibling chains, we see that the complexity of the algorithm is O(|Pertinent Children of Current|). Note that the operation of

Template Q_3 is correct for the Pseudo_Node case, since the sibling chains are maintained with respect to the actual children. Yet the Pseudo_Node is also maintained correctly.

The above algorithmic descriptions, whilst reasonably complete, necessarily omit many programming details. Almost all of them consist of writing the service routines that the algorithms use (procedure Add_to_Circle_Link, procedure Add_Siblings etc.). One detail that is not so straightforward is that of reinitialisation of certain fields. Recall from the beginning of this section that we assume that the nodes in the rest of T except for Pruned (T,S) are empty. This means that we need to reinitialise certain fields back to their original values. For example, a P-node which was full in one particular reduction may not be full for another reduction. This reinitialisation applies to the fields Mark, Data_Label, Partial_Kids, Full_Kids and Full_Kids_Count. It is easily performed by keeping a list of all nodes which are used during the passes. From the reduction pass all full nodes are used and need to be reinitialised, as is Root(T,S). We economise by making the list of used nodes the same structure as the list of full children for a node. Thus, when we are finished with the full children list of a node, we append it via a simple operation onto the list of used nodes. In this way we can dispose of the list of full children efficiently and have a list of nodes to reinitialise easily maintained. The last point to note is, of course, the complexity of the above reduction algorithm.

Theorem 2.11: The Template Matching and Replacement phase (i.e. Pass 2) of the Reduction Algorithm performs in O(|Pruned (T,S)|) steps.
Proof: As discussed above, each template requires O(|Pertinent Children|) steps to execute. Since we never match more than one template to any node, and any node to the same template twice, we must have the required result. ❑

We can summarise the reduction algorithm by the following algorithm.

Algorithm 2.24: Reduction(T,$\mathcal{S}$)

{ Reduce a PQ-tree T so that, for every S $\in$ $\mathcal{S}$, every element

of S appears consecutively in every Frontier(T'),
where T' ≅ T }

```
T = T(𝒰, 𝒰)
   { set T to be the universal tree with respect to 𝒰 }
for each S ∈ 𝒮 do
   begin
      T = Bubble(T, S)
      T = Reduce(T, S)
   end
```

end

At the end of the procedure our PQ-tree T is constrained so that every set $S \in \mathcal{S}$ appears consecutively in Frontier(T). The overall complexity of the procedure is also linear, but to prove this we need to define a few terms first. A *unary* node is a node of Pruned(T,S) which only has one pertinent child. We define the set of all unary nodes in T with respect to S as Unary(T,S). We define the *branching factor* of a node of Pruned(T, S) to be the number of pertinent children of that node in T. We say that a node has an *unary branching factor* if the branching factor of that node is one. Similarly, we say that a node has a *binary branching factor* if the branching factor of that node is two. Next, we redefine the complexity of the algorithm in terms of the size of the set S.

Lemma 2.14: The pruned reduction algorithm requires $O(|S| + |Unary(T,S)|)$ steps to reduce T with respect to S.

Proof: There are only $|S|$ pertinent leaves. Binary branching is the worst non-unary branching factor and this implies that there are at most $O(|S|)$ non-unary nodes in Pruned(T,S). Thus, the total number of nodes in Pruned(T,S) is $O(|S| + |Unary(T,S)|)$ and from Theorems 2.10 and 2.11 the result follows. ❑

Now let us use Lemma 2.14 to generate a result for the overall complexity of our planarity algorithm.

Theorem 2.12: The successive reductions for edges directed into vertices v_k $(2 \le k \le |p|-1)$ may be performed in $O(q + p)$ steps or, equivalently, $O(p)$ steps.

Proof: From Lemma 2.14 we see that we may derive the overall complexity of Algorithm 2.24 as a function of two sums.

The first sum is $\sum_{S \in \mathcal{S}} |S| = O(q)$, since each leaf represents a directed edge.
The second sum is $\sum_{S \in \mathcal{S}} |\text{Unary}(T,S)|$.

We need to show that the total height of our PQ-tree T does not grow unreasonably from repeated template applications. First, note that Root(Pertinent(T,S)) is not a unary node. Also, since we ensure throughout the algorithm that chains are avoided, a unary node may not be labelled full. Thus, we are restricted to considering Templates P_5, P_3 and Q_2, since these templates are the only ones that could possibly apply to unary nodes.
Template P_3 introduces a partial node into Pruned(T,S). Thus, we can apply P_3 at most twice during any one reduction, which is certainly O(p) in total.
Now, for Template Q_2 we consider two sub-cases. Denote by Template Q'_2 the applications of Template Q_2 which do not have a partial node as a child. As for the case for Template P_3, Template Q'_2 may not be applied more than twice during any reduction, and consequently is certainly applied O(p) times throughout the algorithm.
Template Q''_2 we define as the other subcase, where a partial child exists. Templates P_5 and Q''_2 are more tricky to bound. We define Norm(T) to be the number of Q-nodes in T plus the number of nodes in T whose parent is a P-node. We note three points about Norm(T)

(i) Initially Norm(T) is equal to the number of vertices adjacent to v_1.
(ii) No template replacement increases Norm(T) by more than one. Templates P_1, P_4, Q_1, Q'_2 do not affect Norm(T). Templates P_2, P_3 increase Norm(T) by one, but may be applied only O(p) times during the entire algorithm.
(iii) Templates P_6 and Q_3 reduce Norm(T) by one or possibly two during each pass of the algorithm.

(iv) Templates P_5 and Q''_2 reduce Norm(T) by at least one during each pass of the algorithm.

The total applications of all other templates apart from Templates P_5 and Q''_2 are $O(p + q) = O(p)$. Now, consider the modification process that we perform on the tree. Denote by S' the new leaves which we introduce into the tree after a reduction is complete. We certainly do not increase Norm(T) by more than $|S'|$, and so in total by q. We now note, from (i), (ii), (iii) and (iv), that Norm(T) remains $O(p + q) = O(p)$. Furthermore, then the sum $|Unary(T,S)|$ for every $S \in \mathcal{S}$, is also bound by O(p), since, from point (iv), the number of possible applications of Template P_5 and Template Q''_2 is bound, in total, by Norm(T).

Then, using Lemma 2.14 we get that the algorithm is O(p). ❑

Chapter 3

Drawing a Graph

From the results of Chapter 2, we can, in linear time, test if a graph G is planar. In this chapter we are concerned with a planar realisation of a planar graph G. The question arises, for a given planar graph G, what information can we obtain about a planar realisation of G. Such information is of important practical use. As we have already seen, there are many applications for planar graphs. For example, in circuit design theory, as well as testing the circuit design to see if it can be laid out on the plane without wires crossing, it is important to know, once we have established that it can be placed in a planar fashion, how such a layout can actually be described. We restrict the algorithms considered to those with linear time implementations. This is so that we can practically consider drawing graphs with large order (for example, graphs representing VLSI (Very Large Scale Integration) circuits, which often have order > 100 000). Unfortunately, we have the problem that the linear time algorithms are "decidedly opaque" [Rea88].

We present two approaches. The first, which we present in Section 3.1, is a linear algorithm which generates a Rotational Embedding Scheme (RES) for a planar graph. Unfortunately, knowing the RES for G does not immediately lend itself to a drawing of G. Consider, for example, that for graphs G, representing circuits often have size O(100 000) or more. Thus, even knowing the order in which adjacent vertices are embedded around a vertex is of little immediate use if we wish to determine a layout of G.

The second approach, which we present in Section 3.4, is a drawing algorithm which, given a rotational embedding of a graph G, will lay out G, in an aesthetically pleasing manner. Note that, although we consider output to be for the screen, we keep the algorithms generalised, so that a plotter may be used for large graphs. In order to produce a drawing on the plane of a graph G, we need some idea of the acceptable aesthetics for such a drawing. From [Col89] we obtain the following definitions: A

polygon is a closed plane figure bounded by three or more straight line segments which terminate in pairs at the same number of terminal points, and that do not intersect other than at their vertices. Also, a convex polygon is a polygon with no interior angle greater than 180°. An apex of a convex polygon is a terminal point at which the interior angle of the polygon is less than 180°. Chiba, Onoguchi and Nishizeki [CON85] list three desirable characteristics for a drawing of a graph.

(i) All edges are drawn as straight line segments
(ii) A facial cycle is drawn as a convex polygon
(iii) An outer facial cycle of a three-connected component is drawn as a convex polygon.

Consider Figure 3.1, below. To justify the first characteristic, compare Figures 3.1(a) and 3.1(b). Intuitively Figure 3.1(b) is more acceptable, since there is more order in the drawing, it looks "neater". Figure 3.2(c) is a drawing of G with restriction (ii) satisfied as well. When compared to Figure 3.2(b), it may be noted that Figure 3.2(c) is intuitively "neater" in turn than Figure 3.2(b), by virtue of the spacing of the vertices - they are more equitably distributed in terms of distance from their neighbouring vertices.

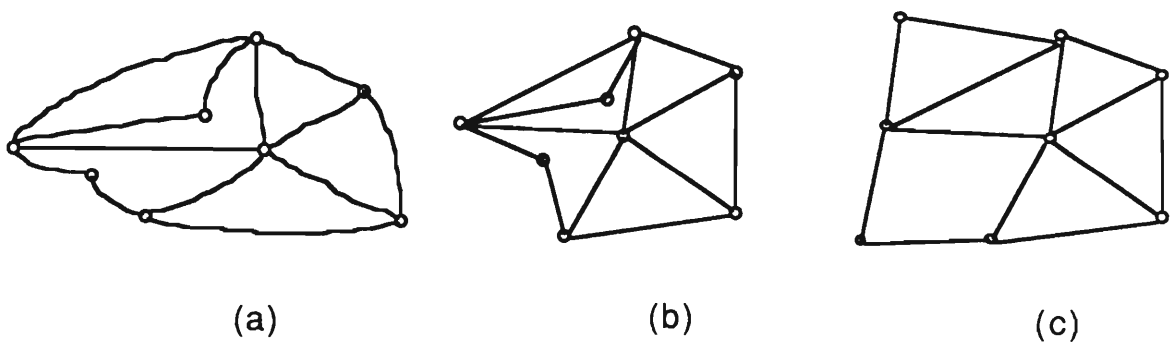


Figure 3.1 - (a) a planar graph G; (b) a drawing of G by straight lines; (c) a convex drawing of G

The third desirable characteristic is that of having the outer cycles of the three-connected components of our graph drawn as convex polygons. Consider the two drawings Figures 3.2(a) and 3.2(b). Although the

drawing in Figure 3.2(b) does not have each facial cycle drawn as a convex polygon, more equitable spacing between neighbouring vertices in a three-connected component is achieved if we sacrifice the criteria that all facial cycles are convex polygons in favour of having all outer facial cycles of 3-connected components drawn as convex polygons. In Sections 3.2 and 3.3, we present linear algorithms for determining the 3-connected components of a planar graph.

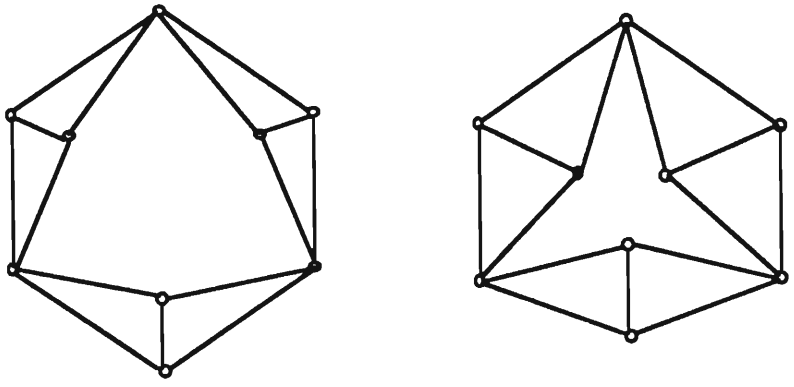


Figure 3.2 - (a) a convex drawing of a graph G; (b) a modified drawing of G

In Section 3.4, we shall present two linear algorithms, by Chiba, Yamanouchi and Nishizeki [CYN84], that produce a reasonably satisfactory drawing of a 2-connected graph G which satisfies characteristics (i) and (ii). The problem of incorporating the third characteristic into the drawing algorithm is discussed in [CON85]. It is based on the two algorithms we are going to present, and differs only in that it draws the graph "piecemeal" by repeatedly calling the drawing algorithm presented in [CYN84].

We define a convex drawing of a planar graph G to be a drawing of G on the plane so that all edges are represented by straight lines, with no two lines intersecting, except possibly at their endpoints, vertices are represented by terminal points, and each interior region boundary of the drawing is a convex polygon.

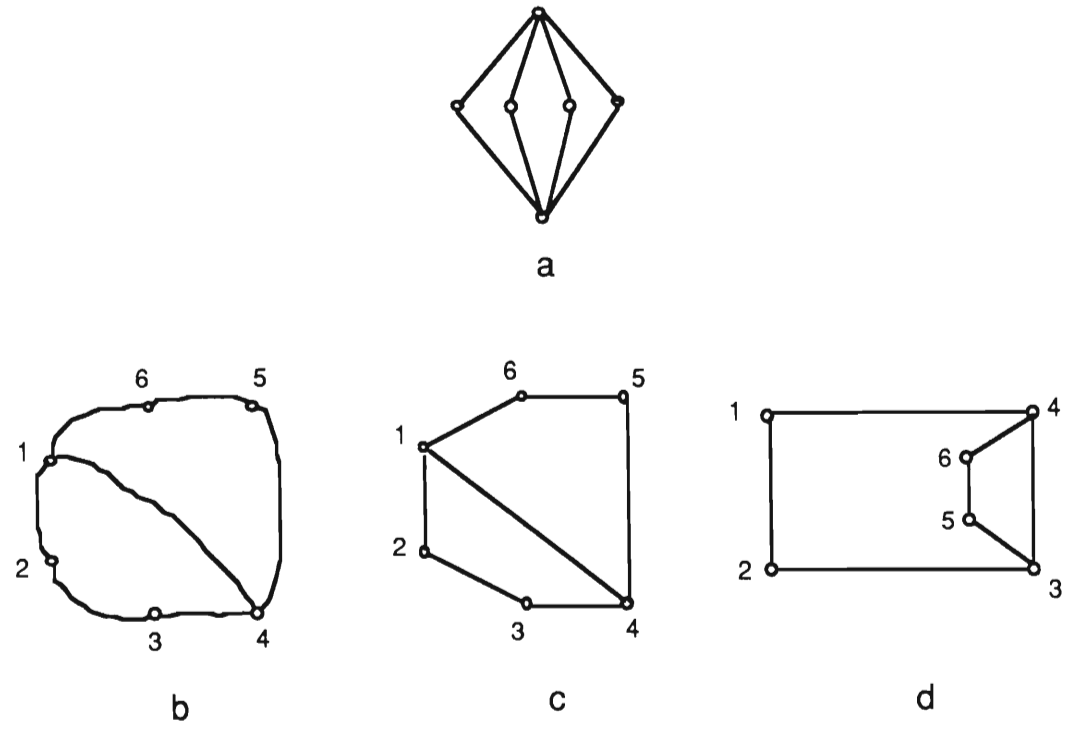


Figure 3.3 - (a) a graph with no convex drawing
(b) a planar graph G; (c) a convex drawing of G; (d) another drawing of G

Not every graph has a convex drawing. The graph in Figure 3.3(a) does not have a convex drawing. The choice of outer facial cycle is important in determining a convex drawing of a graph. Consider the graph G in Figure 3.3(b). If we choose the outer facial cycle to be 1, 2, 3, 4, 5, 6, then we can obtain the convex drawing of G, illustrated in Figure 3.3(c). However, if we choose the outer facial cycle of our drawing to be 1, 2, 3, 4; then we cannot obtain a convex drawing of G. From this motivation, we define a convex polygon S^* of a facial cycle S of a plane graph G to be *extendible* if there exists a convex drawing of G having S^* as the outer facial cycle of G. A facial cycle S is extendible if and only if S has an extendible convex polygon S^*.

Given a planar graph G and an extendible convex polygon S^* of a facial cycle S, the first algorithm, Algorithm Convex_Draw, will provide a convex drawing of a planar graph G in linear time. The second algorithm, Algorithm Convex_Test, tests if G has a convex drawing, and, if so, outputs the extendible convex polygon S^* of a facial cycle S.

Let {a, b} be a pair of vertices in a 2-connected multigraph G. Suppose the edges of G are partitioned into sets E_1, E_2, ..., E_n such that every two edges which lie on a common path, which does not contain any vertex of {a, b} except as endpoints are in the same class. The classes E_i are called *separation classes* of G with respect to {a, b}. If there are at least two separation classes, then {a, b} is a *separation pair* unless there are exactly two classes and one contains a single edge, namely the edge e = ab. If G is a 2-connected graph such that no separation pair {a, b} exists, then G is *3-connected*.

Algorithm Convex_Test requires some knowledge about the separation pairs in G. Thus, before presenting the testing and drawing algorithms, we need to consider linear time triconnectivity testing algorithms. There are two such algorithms. The first is based on the planarity testing algorithm by Hopcroft and Tarjan covered in Section 2.3. The second algorithm is based on the planarity testing algorithm using PQ-trees, covered in Sections 2.4, 2.5 and 2.6.

Thus, in Section 3.1 we present a linear time embedding algorithm, in Sections 3.2 and 3.3 we present two linear time triconnectivity algorithms. Finally, in Section 3.4 we present linear time drawing and testing algorithms.

Section 3.1

Finding an Embedding of a planar graph

We use the PQ-tree algorithm given in sections 2.3, 2.4 and 2.5 to modify the planarity testing algorithm to also give an embedding of the graph. The algorithm is by Chiba, Nishizeki, Abe and Ozawa [CNAO85]. We follow Chiba and Nishizeki [CN88] for details.

Consider the following algorithm to generate an embedding using the Lempel, Even and Cederbaum [LEC67] planarity testing algorithm. We may use the Bush Forms discussed in Section 2.3 to obtain the embedding directly as we generate the Bush Forms. Algorithm 3.1, below, gives an outline of such an algorithm.

Algorithm 3.1: First_Embedding

Generate first Bush Form $\hat{B}_1$

Write down the corresponding embedding by reading the frontier of $\hat{B}_1$

For k = 2 to p do

Generate Bush Form $\hat{B}_k$

Adjust the adjacency lists by reading frontier of $\hat{B}_k$

end

Thus, as we have already seen in Theorem 2.9, modifying the PQ-tree data structure to implement Algorithm 3.1 and reading the frontier of the PQ-tree after every reduction is possible. However, Algorithm 3.1 is not of linear complexity, since we read the entire frontier of $\hat{B}_k$, and we adjust the adjacency lists after every reduction. The linear time algorithm we present is based on Algorithm 3.1, but we avoid the overhead of having to adjust the adjacency lists after every reduction.

Consider a 2-connected, st-numbered, planar graph G. We direct the edges from vertices of lower st-number to vertices of higher st-number, and obtain a digraph D. We define an *upward embedding* A_u of G to be an

embedding of D. Thus, in an upward embedding A_u of G, an adjacency list of a vertex v, with st-number k, will only contain vertices adjacent to v in D with st-number j (j > k). Thus, in an upward embedding of D, we represent D by the adjacency list of the in-neighbourhood of its vertices.

The algorithm runs in two passes. Suppose we wish to find an embedding of an st-numbered 2-connected, planar graph G. The first pass finds an upward embedding A_u of G. The second pass extends the embedding A_u of G to an embedding A of G. We will discuss the second pass first, since it is conceptually clearer and briefer, and justifies the approach used in the first pass.

Lemma 3.1: Let A be an embedding of an st-numbered, planar (p, q) graph G. Then, for any vertex $v \in V(G)$, all vertices $u \in V(G)$ adjacent to v with lower st-number than v are grouped consecutively in the adjacency list of v in A.

Proof: See Figure 3.4. Consider the Bush Form $\hat{B}_k$, where k is the st-number of v. If all the edges incident to v and from vertices having lower st-number than k are not consecutive in the adjacency list of v in A, then, from Section 2.3, we have that either Lemma 2.11 or Theorem 2.7 are contradicted. ❑

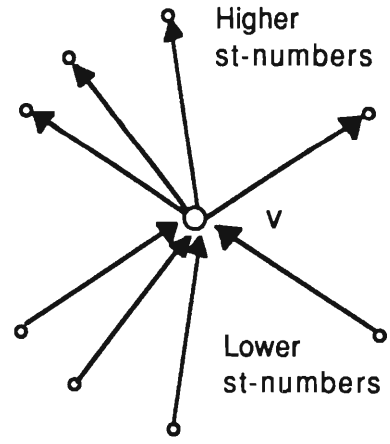


Figure 3.4 - Embedding of edges about v according to st-numbers

Now, the first pass of the algorithm will find an upward embedding A_u of G. From Figure 3.4, we observe that the extension of the upward embedding A_u to an embedding A of G involves adding the extra edges, in order of appearance around v, which are incident with vertices of greater st-number in the adjacency list of v. We say that an upward embedding A_u of G is *extended* to an embedding A of G if, for every vertex $v \in V(G)$, all edges incident with vertices with lower st-number than the st-number of v, appear consecutively in the adjacency list of v, and in the same order as they appeared in the adjacency list of v in A_u. If A is an embedding of G, then, for a vertex $v \in V(G)$, we denote the adjacency list of v in A by A(v).

Algorithm 3.2, shown below, will extend an upward embedding A_u of G to an embedding A of G by performing a Depth-First Search (DFS) on G. As we shall show, the resulting adjacency structure A is indeed an extended embedding of G.

Algorithm 3.2: Extend_Embedding

```
{ Start the copy by copying known edges }
Copy all adjacency lists of A_u to A

{ initialise for a DFS }
For every vertex v ∈ V(G)
    Mark(v) = New
For every edge e ∈ A_u do
    Mark(e) = Unscanned

{ vertex v with st-number p is the root of the DFS tree}
Current = p
Parent(Current) = ∅

while Current ≠ ∅ do
                                { Note part of the DFS tree }
    Mark(Current) = Old
    if there is an edge unscanned in Adjacency list A_u of Current
        then
                                { Check unchecked edge }
            Let the first such unscanned edge be e = v → Current
            Mark(e) = Scanned
            Add Current to head of adjacency list A(v)
                                { Check possible new descendant }
            if Mark(v) = New
                then
                                { Add to DFS tree the edge e }
```

```
                    Parent(v) = Current
                    Current = v
                else
        else                        { all done so backtrack to parent }
            Current = Parent(Current)
```

end

It is not hard to see that Algorithm 3.2 does perform a DFS, builds a DFS tree T, and is, for planar graphs, of complexity O(p). Also, since Algorithm 3.2 does perform a DFS, and G is st-numbered, every edge in the digraph D of G is scanned, since there is a path in digraph D of G from t to every other vertex of D. Thus, all edges not in A_u, but in G, are added to build the adjacency structure A. Thus, A contains every edge of E(G). To prove that A is an extended embedding of the upward embedding A_u of G, we have the following lemma.

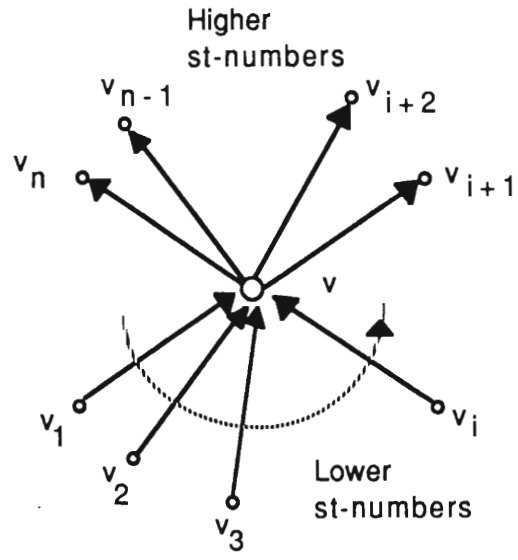


Figure 3.5 - Embedding around v according to st-numbers

Lemma 3.2: Algorithm 3.2 extends an upward embedding A_u of G to an embedding A of G.

Proof: Consider a vertex v, with st-number k. See Figure 3.5. Suppose that the vertices adjacent to a vertex v, with lower st-number than k, appear in the order v_1, v_2, ..., v_i, in $A_u(v)$. From Lemma 3.1, we know that all vertices with greater st-number than k appear after all vertices of lower st-number than k in A. Let the vertices with greater st-number be v_{i+1},

v_{i+2}, ..., v_n. In Algorithm 3.2, we observe first that we copy the upward embedding A_u to A. Then, we add each vertex Current which was incident to an edge v $\rightarrow$ Current of A_u, onto the adjacency list of v. Thus, for A to be an embedding, the edges incident from v are scanned in the order $v_{i+1}v$, $v_{i+2}v$, ..., v_nv. Then, by adding the edges in the order v_nv, $v_{n-1}v$, ..., $v_{i+1}v$ to the front of the adjacency list of v, we shall obtain the embedding arrangement depicted in Figure 3.5.

Suppose that this is not the case, and that there are two edges v_pv and v_qv such that, in the algorithm, they are scanned in the order v_qv and v_pv, even though v_qv appears before v_pv as we proceed anticlockwise around v in the embedding of D. See Figure 3.6. Let P_q be the path, in the DFS tree T, from t to v_q, and let P_p be the path, in the DFS tree T, from t to v_p. Suppose w is the last vertex that P_p and P_q have in common, and that $wv'_p \in E(P_p)$ and $wv'_q \in E(P_q)$. Note that, since the paths P_p and P_q were from t, both the st-number of v'_q and the st-number of v'_p are less than the st-number of w. Consequently, both edges appear in $A_u(w)$. Because A_u is an upward embedding and because of the choice of v'_q before v'_p, wv'_q appears before wv'_p in $A_u(w)$. We let P'_p be the path w, v'_p, ..., v_p and P'_q be the path w, v'_q, ..., v_q. Consider now the cycle C formed by proceeding along the path P'_p, along the edges v_pv and vv_q and along the path P'_q. All the vertices in $A_u(v)$ must lie in the interior of the cycle, or else Lemma 3.1 would be contradicted. Since the vertex s (with st-number 1) is on the exterior region of the embedding (because it is only an upward embedding), v is not s. But then, by the definition of an st-numbering, there exists a path from s to v of vertices with st-number less than or equal to k. However, all vertices on C have st-number greater than k. We therefore have a contradiction, and we must encounter the edges incident to v in the correct order. ❑

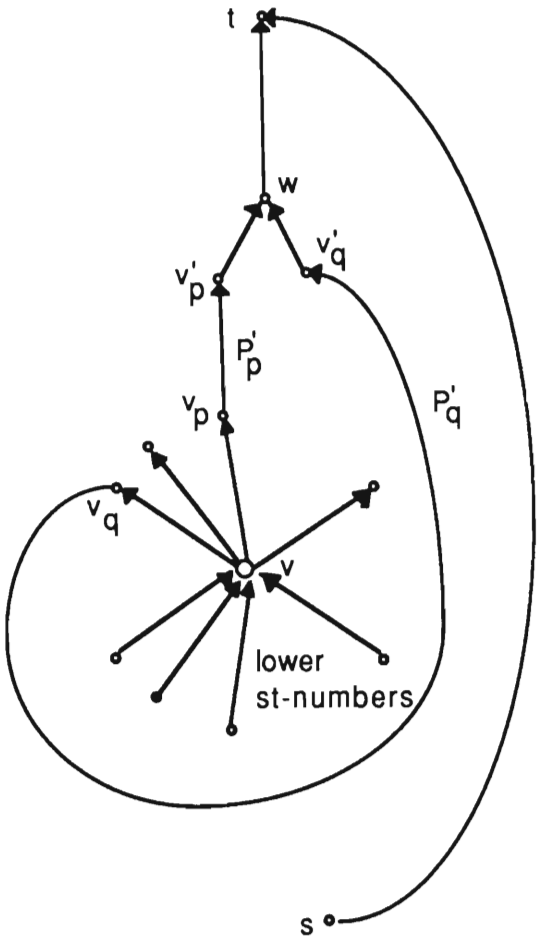


Figure 3.6 - Proof of Lemma 3.2

Thus, Algorithm 3.2 gives a simple technique for extending an upward embedding A_u of G to an embedding A of G. This algorithm is important, since it indicates that it is sufficient, when reducing the PQ-tree, to obtain, for each vertex $v \in V(G)$, $A_u(w)$ from the pruned pertinent subtree Pruned(T, S). We may then, in linear time, extend the upward embedding so obtained, to an embedding of G.

We may now turn our attention to the first pass of the algorithm. From Section 2.4 and Section 2.3, we can see that, for a vertex v with st-number k, by reading the frontier of the pruned pertinent subtree after we have performed the k-th reduction, we have an upward embedding for the vertex v, up to reflection. That is, the order in which the vertices appear in the adjacency list of v in the embedding remain the same as they were

when read after the reduction, except for possibly a reflection. Now, consider the remaining reduction operations on the PQ-tree.

During permutations about cut-vertices, the order will remain the same. However, during reflections of v-blocks the order might be reversed. We define each reflection operation to be a *reversion*. Thus, by counting the number of reversions of the PQ-tree node representing v, one may obtain, for each vertex v, the correct embedding $A_u(v)$, by simply reversing $A_u(v)$ if the number of reversions is odd. The problem is that a simple algorithm to count the number of reversions will not be linear. Also, the node representing v may disappear from the tree, if, for example, the relevant node is part of a full node during another (later) reduction. Chiba, Nishizeki, Abe and Ozawa [CNAO85] use a very elegant solution to avoid the problems associated with counting the number of reversions.

First, consider the problem of scanning the frontier of the pruned pertinent subtree, Pruned(T, S), after reduction, to produce $A_u(v)$ for the upward embedding A_u. To read the frontier of Pruned(T,S), we perform a DFS. For the purposes of the DFS, we redefine an endmost child to be a full child with only one full immediate sibling. The DFS proceeds by selecting Root(T, S) as the start vertex. Then, the DFS proceeds down Pruned(T, S) in the following manner. If the current node is a leaf node, then add the edge it represents to $A_u(v)$. If the current node is a Q-node, then we select an endmost node, perform a DFS on that node and, by traversing the immediate sibling chains, move through all the full children, performing a DFS on each in turn (i.e. perform a DFS on the subtree rooted at that child), before returning to the Q-node. If the node is a P-node, then, in any order, perform a DFS on each child and return to the P-node. Algorithm 3.3 gives the full DFS. Initially we call the algorithm with Root(T, S).

Algorithm 3.3: Read_Pertinent_Frontier (Current)

```
Case Current of

{ walk in order through full kids and DFS on each }
   Q-Node
```

```
                Previous = ∅
                Temp = an endmost child of Current
                repeat
                    Read_Pertinent_Frontier (Temp)        { recurse }
                    Get_Next_Sibling (Previous, Temp)     { and get next sib }
                until Temp not Pertinent

            { perform DFS on all the kids, in any order }
                P-Node
                    For each Child of Current do
                        Read_Pertinent_Frontier (Child)

            { add to the adjacency list }
                Leaf
                    Add_to_Adjacency_List (Current, v)
end
```

Notice that, when traversing the immediate sibling fields, we need to know the previous node and the current node. The other sibling is then given by examining the immediate sibling field and choosing the neighbour node not the same as the node given by Previous. It is easy to see that the number of operations to read the frontier of Pruned(T, S) in Algorithm 3.3 is O(|Pruned(T, S)|).

It is impossible for the algorithm, at this stage, to say whether the adjacency list obtained for a vertex v from Algorithm 3.2 is $A_u(v)$, or whether it is a reversion of $A_u(v)$. For example, we may read Frontier(Pruned (T, S)) from left to right, but in the final embedding, we have reflected the node, and thus $A_u(v)$ is a reversion. We need to note the direction we took when reading Frontier(Pruned(T, S)). Consider Figure 3.7(a), below. The only "direction" that we can indicate is related to the empty siblings on either side of the pertinent children. Although in Figure 3.7(a) we may easily note an ordering (for example, say, left to right), it is impossible for the algorithm to determine this without going outside Pruned(T, S). If this direction is opposite to the final order in which the edges appear when we embed the graph, then $A_u(v)$ is a reversion.

To this end, we insert an indicator node into the sibling list. See Figure 3.7(b). The indicator gives a direction, relative to the empty siblings, that the Frontier(Pruned (T, S)) was read in. The indicator holds two values:

the "left" and "right" empty neighbours of Pruned(T, S). Now, when further reductions cause the Q-node parent to be reflected, the reversion is implicitly noted by the indicator. When we read the indicator when reading Frontier(Pruned(T, S)) at a later stage, and the left and right siblings differ from the order in which we are reading Frontier(Pruned(T, S)), then we know that $A_u(v)$ is a reversion of the order of $A_u(v)$ at this stage. We still cannot tell the correct order in the final solution.

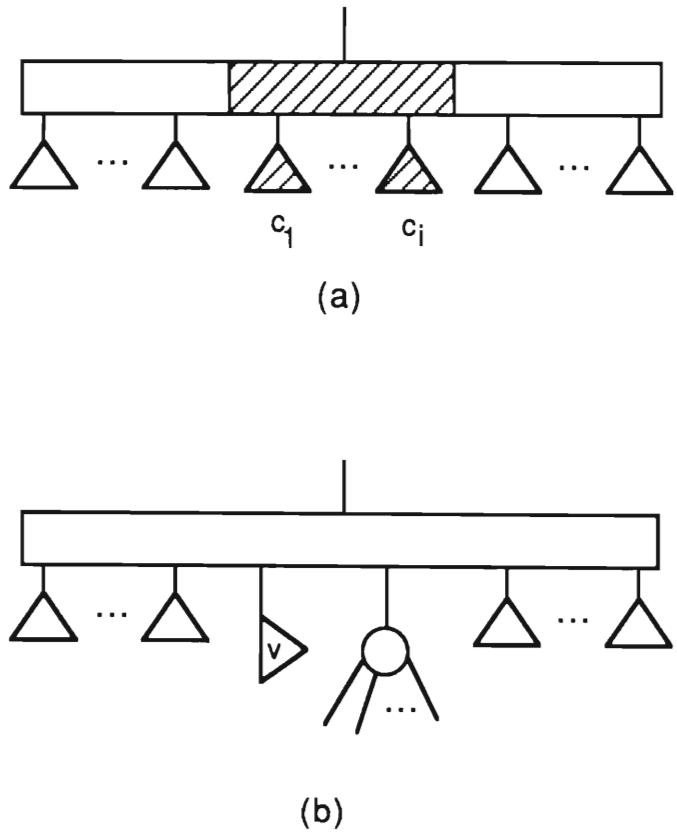


Figure 3.7 - Pertinent Children and Direction Indicators

During the template matching pass, we ignore the indicators. When we access the immediate siblings of a neighbour, we ignore the presence of indicators and access the next PQ-node instead. However, when merging Q-node sibling lists (for example in Template Q_3), we necessarily treat the indicator as part of the sibling list. Note also that all indicators are leaves.

Thus, the template algorithms in Section 2.5 remain the same. The only section of code which changes is, of course, the vertex addition step. We

note that if Root(T, S) is full, then the order of the edges in an embedding is allowed to be the same as read by Algorithm 3.3. A reflection of the ordering in a parent node may reverse the order in which Root(T, S) and its siblings appear, but the ordering of the children of Root(T, S) is totally unrelated. Thus, at this stage in the algorithm, if there are any indicators in Pruned(T, S), then we may fix their ordering permanently. Algorithm 3.4 gives the modified vertex addition step of the reduction algorithm, Algorithm 2.11.

Algorithm 3.4: Modified_Vertex_Addition (Root, v)

```
{ Root is Root(T, S) and v is the vertex we are reducing for }

Let ℓ1, ℓ2, ..., ℓj be the full leaves and
let i1, i2, ..., ik be the indicators
scanned, in that order, using Algorithm 3.3

Au(v) = { ℓ1, ℓ2, ..., ℓj)                    { set the adjacency list up }

{ Adjust indicators first }
If Root is not Full
    then                    { new direction indicator to be inserted }
        Add indicator i as a child of Root at arbitrary position,
            directed from ℓ1 to ℓj
        Add indicators i1, i2, ..., ik as kids of Root at arbitrary positions
    else                    { all indicators i1, i2, ..., ik are now determined }
        For each indicator im do
            if im is directed from ℓj to ℓ1
                then reverse Au(im)
        Delete i1, i2, ..., ik              { they are now redundant }
    end

{ Now the normal vertex addition step }
if Root is not full
    then
        Replace sequence of full children and their descendants by a
            P-node with children the edges directed out of v
    else
        Replace Root(T,S) and its descendants by a P-node with
            children being edges directed out of v
end
```

Notice that once we have Root(T, S), which is labelled full, then the adjacency lists represented by indicators within Pruned(Root(T, S)) are also determined.

Lemma 3.3: Algorithm 3.4 obtains an upward embedding A_u of a given planar graph G.

Proof: We note that, for each vertex $v \in V(G)$ with st-number k, $A_u(v)$ contains all leaves ℓ_i with st-number less than k. Also, from Algorithm 3.3, the order in which they appear is either clockwise or anticlockwise. Thus, we show that for every $A_u(v)$, the edges in $A_u(v)$ appear in clockwise order. We have two cases, depending on whether the indicator was inserted into the PQ-tree T, or not.

If an indicator was not inserted into T, then Root(T, S) is full and, as already discussed, the order of the children remain as found by Algorithm 3.3. Thus, the order of the edges may be considered to be clockwise.

For the case when the indicator is inserted into T, we note that when the algorithm finishes, Root(T, S) is a P-node. From Algorithm 3.3, we will scan the entire tree, and thus all indicators are deleted from T at some stage during the algorithm. Assume that, for a vertex v, the indicator i is deleted after reduction for a vertex w is complete. We may now look at the relative direction of indicator i. If the leaves scanned by Algorithm 3.3 for vertex w are scanned in the same direction as indicator i, then $A_u(v)$ is correct. If the indicator disagrees, then we have two cases. Either, there were an odd number of reversions and the edges were scanned in the same order, or there were an even number of reversions and the edges were scanned in opposite order. In either case, by reversing the order of $A_u(v)$, we may obtain the correct adjacency list. Lastly note that, for the vertex w, from Algorithm 3.4, the Root(T, S) is full, and so $A_u(v)$ and $A_u(w)$ are never changed afterwards. Thus $A_u(v)$ remains in the correct order. ❑

We note that Algorithm 3.4 is not linear. This overhead in complexity occurs because of the re-insertion of indicators back into the tree. In the worst case we may have to re-insert each indicator back into the tree O(p) times, and since there are O(p) indicators, we derive an overall complexity of reinserting the direction indicators of $O(p^2)$. The linearity of Algorithm 3.4 is, however, easily accomplished.

Notice that the re-insertion of the indicators i, after a reduction for a vertex v, into the tree is to keep track of the number of reversions for their corresponding lists $A_u(w)$. Consider any indicator i, for a vertex w, which has to be re-inserted into T. If we note, before we re-insert i, whether the lists $A_u(v)$ and $A_u(w)$ were read in opposite orders, then we may simulate the insertion of i as follows. Denote by *Conflict* a boolean variable which is true if $A_u(v)$ and $A_u(w)$ were read in opposite orders. We then insert an element into $A_u(v)$ which stores the variable Conflict and the vertex w. Now, when the order of $A_u(v)$ is decided, then we decide the order of $A_u(w)$. We have two cases.

If $A_u(v)$ has to be reversed, then if Conflict is true, then $A_u(w)$ was read in the correct order. But, if Conflict is false, then the adjacency lists were read in the same order, and so $A_u(w)$ has to be reversed as well.

If $A_u(v)$ does not have to be reversed, then the conditions are an exact opposite of the first case. If Conflict is true, then $A_u(w)$ was read in opposite order, and so has to be reversed. If Conflict is false, then the adjacency lists were read in the same order, and so, neither does $A_u(w)$ have to be reversed.

We note that there may be other Conflict variables in the list $A_u(w)$, and so we repeat the procedure for them. For indicator i encountered in Algorithm 3.3, we can easily modify Algorithm 3.3 to determine the variable Conflict, and insert an element containing the variable Conflict and the vertex the indicator represents into $A_u(v)$. Then, in Algorithm 3.4, when Root(T, S) of the current reduction is full, we call a procedure Correct_Adjacency_Lists (v, No_Reverse) which corrects $A_u(v)$ and its dependant adjacency lists $A_u(w)$, given that we do not want to reverse $A_u(v)$. Algorithm 3.5 presents the modified version of Algorithm 3.4.

Algorithm 3.5: Modified_Vertex_Addition (Root, v)

{ Root is Root(T, S) and v is the vertex we are reducing for }

Let $f_1, f_2, \ldots, f_j$ be the full leaves and indicators scanned using Algorithm 3.3

```
A_u(v) = ( f_1, f_2, ..., f_j)

{ Now the normal vertex addition step }
if Root is not full
    then
        Replace sequence of full children and their descendants by a
            P-node with children the edges directed out of v
    else
        Replace Root(T,S) and its descendants by a P-node with
            children being edges directed out of v
        Correct_Adjacency_Lists (v, No_Reverse)
```

end

Algorithm 3.6 gives the procedure Correct_Adjacency_Lists (v, Reverse) to update the adjacency list of a vertex v, given the boolean variable Reverse, which indicates if $A_u(v)$ must be reversed or not.

Algorithm 3.6: Correct_Adjacency_Lists (v, Reverse)

```
if Reverse
    then
        For each element e ∈ A_u(v)
            if e is an indicator
                then                        { recursively check lists }
                    Correct_Adjacency_Lists (w,  not Conflict)
                else                        { normal edge }
                    Add_to_Stack (e)
        A_u(v) = ∅
        For each element e on Stack             { add in reverse order }
            Pop(e)
            Add e to A_u(v) at head of list
    else                                    { do not reverse }
        For each element e ∈ A_u(v)
            if e is an indicator
                then                        { recurse and delete }
                    Correct_Adjacency_Lists (w,  Conflict)
                    Delete from A_u(v)
```

end

Lemma 3.4: The complexity of Algorithms 3.5 and 3.6 is O(p).

Proof: Certainly, the complexity of Algorithm 3.5 is unchanged from the original vertex addition algorithm given in Algorithm 2.11, and therefore has O(p) complexity overall. Note that, in an analogous observation to that in Lemma 3.3, each indicator is added into an adjacency list, or if not,

then we call Algorithm 3.6 explicitly. Thus, Algorithm 3.6 will only be called once for each vertex, and therefore O(p) in total. ❑

Algorithm 3.7, finds, for a 2-connected planar graph G, an embedding A.

Algorithm 3.7: Embedding (G, A)

{ Find an embedding A of G }

```
𝒰 = set of edges directed out of v1   { initial universal set }
T = Universal tree of 𝒰               { the initial tree }
Au(1) = ∅
for Current = 2 to |p| do             { for each vertex }
    S = set of edges directed into vertex Current      { set to reduce }
    T = Reduce(T,S)                                    { do reduction }
    S' = set of edges directed out of vertex Current   { new leaves }
    Modified_Vertex_Addition (Root, Current)           { add them }
    𝒰 = (𝒰 - S) ∪ S'                                   { update universal set }

Extend_Embedding        { extend upward embedding }
```

end

Notice that in the upward embedding, v_1 has no incident edges. Also, the last pass of the algorithm is for the vertex v_p. We have the following result.

Theorem 3.1: Algorithm 3.7 correctly generates an embedding A of G, and has complexity O(p).

Proof: From Lemma 3.3 and Theorem 2.12, Algorithm 3.7 runs in linear time. From the discussion preceding Algorithms 3.5 and 3.6, an upward embedding A_u of G is found correctly. Lastly, from Lemma 3.2, Algorithm 3.7 correctly extends the upward embedding A_u of G to an embedding A of G. ❑

We close this section with an example of the embedding process. Figure 3.8 shows a graph that we will use as an example.

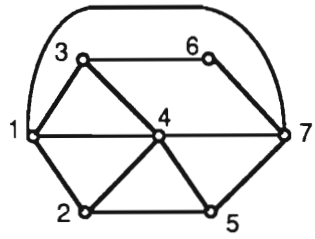


Figure 3.8 - An example st-numbered graph

For the example, when we refer to an st-number, we are implicitly referring to the corresponding vertex. Initially, $A_u(1) = \{\ \}$. The PQ-tree reduction pass proceeds trivially for vertices 2 and 3, and $A_u(2) = \{1\}$ and $A_u(3) = \{1\}$. Figure 3.9(a), below, shows the PQ-tree so far. Figure 3.9(b) shows the PQ-tree after reduction for vertex 4 is complete.

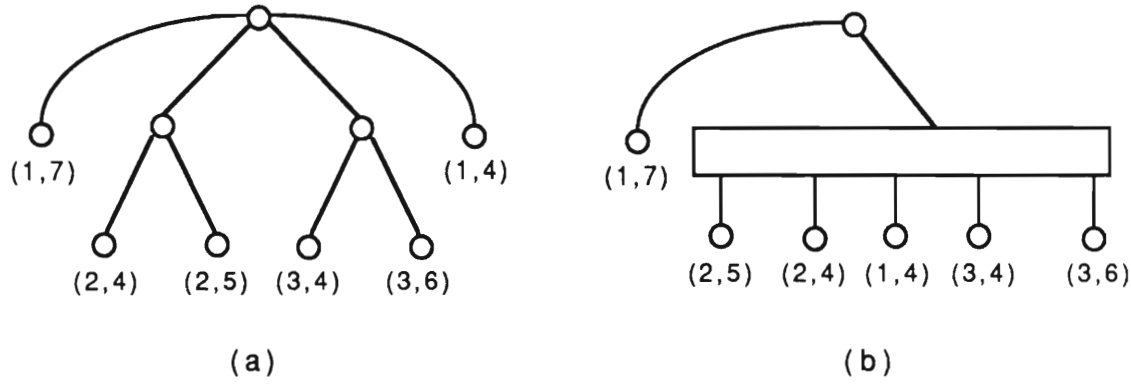


Figure 3.9 - Reduction pass before and after reducing for vertex 4

Figure 3.10 shows the tree after the vertex addition stage for vertex 4 is complete. We now have that $A_u(4) = \{3, 1, 2\}$. We insert an indicator reflecting the direction that we took to read the frontier of Pruned(T, S), and we insert a new P-node in place of all the pertinent children.

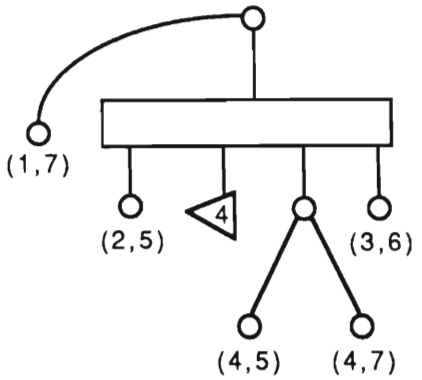


Figure 3.10 - After completion of vertex addition step for vertex 4

We continue the reduction pass, and reduce for vertex 5. This time, there is an indicator in the frontier of Pruned(T,S). We add the indicator to $A_u(5)$, and get $A_u(5)$ = {2, {indicator 4, Conflict}, 4}. Lastly, we add the single leaf edge (5, 7) and indicator. Figure 3.11(a), below, shows the reduction pass after reducing for vertex 5, and Figure 3.11(b) shows the reduction pass after the vertex addition for vertex 5 is complete.

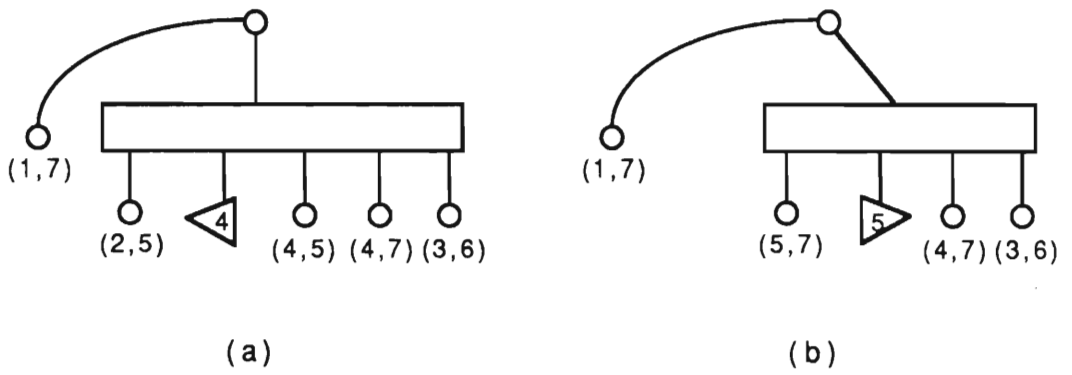


Figure 3.11 - (a) After reducing for vertex 5; (b) after vertex addition for vertex 5

Reduction for vertex 6 is trivial, and we get $A_u(6)$ = {3}. Reduction for vertex 7 is also easy, and we get $A_u(7)$ = {1, 5, {indicator 5, no Conflict}, 4, 6}. Therefore, the final upward embedding created by the algorithm is.

$A_u(1) = \{\}$
$A_u(2) = \{1\}$
$A_u(3) = \{1\}$
$A_u(4) = \{3, 1, 2\}$

$A_u(5)$ = {2, {indicator 4, Conflict}, 4}
$A_u(6)$ = {3}
$A_u(7)$ = {1, 5, {indicator 5, no Conflict}, 4, 6}

Since Root(T, S) is full, we call Algorithm Correct_Adjacency_Lists for vertex 7. The only adjacency list that has to be reversed is $A_u(4)$. The corrected upward embedding is given below.

$A_u(1)$ = { }
$A_u(2)$ = {1}
$A_u(3)$ = {1}
$A_u(4)$ = {2, 1, 3}
$A_u(5)$ = {2, 4}
$A_u(6)$ = {3}
$A_u(7)$ = {1, 5, 4, 6}

A simple DFS on the upward embedding digraph will yield the correct embedding A for G. Embedding A is shown below.

A(1) = {3, 4, 2, 7}
A(2) = {4, 5, 1}
A(3) = {6, 4, 1}
A(4) = {7, 5, 2, 1, 3}
A(5) = {7, 2, 4}
A(6) = {7, 3}
A(7) = {1, 5, 4, 6}

In the next two sections we shall present two linear time triconnectivity algorithms, which are used to produce a linear time drawing algorithm which accepts as input an embedding from Algorithm 3.7.

Section 3.2

Hopcroft and Tarjan Triconnectivity Testing by Path Addition

The algorithm by Hopcroft and Tarjan [HT73b] is based on the same ideas that their planarity testing algorithm of Section 2.2 was based on. The presentation we follow in this section is the original approach by Hopcroft and Tarjan [HT73b]. Their original algorithm dealt with multigraphs. For our planarity testing algorithms, we can restrict our attention to planar (p, q) graphs. Before we introduce the algorithm, we need a few definitions and concepts. We use the approach by Hopcroft and Tarjan [HT73b] for details. We repeat, for convenience, the definitions given at the beginning of the chapter.

Let {a, b} be a pair of vertices in a 2-connected multigraph G. Suppose the edges of G are partitioned into sets E_1, E_2, ..., E_n such that two edges belong to the same E_i, if and only if there is a path which contains them, and whose internal vertices are different from a and b. The classes E_i are called *separation classes* of G with respect to {a, b}. If there are at least two separation classes, then {a, b} is a *separation pair* unless there are exactly two classes and one contains a single edge. If G is a 2-connected graph with no separation pair {a, b}, then G is *3-connected*.

Let {a, b} be a separation pair of G. Let the separation classes of G with respect to {a, b} be E_1, E_2, ..., E_n. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{n} E_i$ such that $|E'| \geq 2$ and $|E''| \geq 2$. Let $G_1 = (V(E'), E' \cup \{a, b\})$ and let $G_2 = (V(E''), E'' \cup \{a, b\})$. The graphs G_1 and G_2 are called the *split graphs* of G with respect to {a, b}. Replacing G by the split graphs G_1 and G_2 is called *splitting* G. We note that there may be many ways of splitting a graph G, even with respect to the same separation pair. With every splitting operation, we assign a unique label to distinguish that splitting operation from others. The new edges e = ab which we add to E' and E'' to obtain

$E(G_1)$ and $E(G_2)$ are called *virtual edges*, and they are assigned the same label as the splitting operation. Note that the virtual edges referred to in this and the next section should not be confused with the virtual edges which were used in connection with bush forms. If the splitting operation at pair {a, b} has label i, then the virtual edges are written (a, b, i). If G is 2-connected, then so is any split graph.

If we repeat the process of splitting split graphs until there are no more separation pairs, we note that the remaining graphs are triconnected. We call these graphs the *split components* of G. They are not necessarily unique. It is important to know that there is a bound on the number of edges in the split components of G.

Lemma 3.5: Let G be a 2-connected graph on q edges, with split components $G_1, G_2, ..., G_m$. Further, suppose that the order of the split components are $q_1, q_2, ..., q_m$. Then, $q_1 + q_2 + ...+ q_m \leq 3q - 6$.
Proof: We proceed by induction on the number of edges in G. If G has three edges, then the result is proved because the graph cannot be split. Suppose the lemma is true for graphs with k edges, $3 \leq k \leq q-1$ edges. Let G be a graph with q edges. If the graph G cannot be split, the result is immediate again. If the graph can be split, then suppose each split graph G_1 and G_2, contain q_a and q_b edges respectively, where $q_a + q_b = q + 2$. By the inductive hypothesis, both the split graphs satisfy the lemma. Thus, the total size of the split components is $q' \leq 3.q_a - 6 + 3.q_b - 6 = 3(q + 2) - 12 = 3q - 6$. ❑

Therefore, the number of times we perform the splitting process is O(q). It remains to find a O(q) algorithm to find the separation pairs, and the split components. A further problem arises because the split components of G are not unique. However, we may partially reassemble the split components to obtain "unique components".

Consider the following operation. Suppose two split graphs, G_1 and G_2, contain the same virtual edge (a, b, i). Consider a new graph H formed by a "merging operation", where $V(H) = V(G_1) \cup V(G_2)$ and $E(H) = E(G_1) \cup E(G_2)$ - (a, b, i). Then, H is called the *merge graph* of G_1 and G_2, and the

process of creating H is called *merging*. If we carry out the merging process on the split components of G, then eventually we will obtain the original graph G. There are three types of split components, namely triple bonds, triangles and 3-connected components. Triple Bonds are three edges joining the same pair of vertices. Triangles are of the form a, b, c, a. Lastly, 3-connected components are split components with more than four edges and no separation pairs. Consider merging all triangles together to give a set $\mathcal{R}$ of rings, and all triple bonds together to give a set $\mathcal{B}$ of bonds. If $\mathcal{T}$ is the set of 3-connected components, then the set of graphs $\mathcal{R} \cup \mathcal{B} \cup \mathcal{T}$ gives the set of triconnected components of G. We have the following lemma from [Mac37] and [TH72].

Lemma 3.6: The set of triconnected components of G are unique.

To illustrate these concepts, consider the following example of a graph G with split components G_1, G_2, ..., G_5. We indicate the virtual edge i by the label $v\,i$.

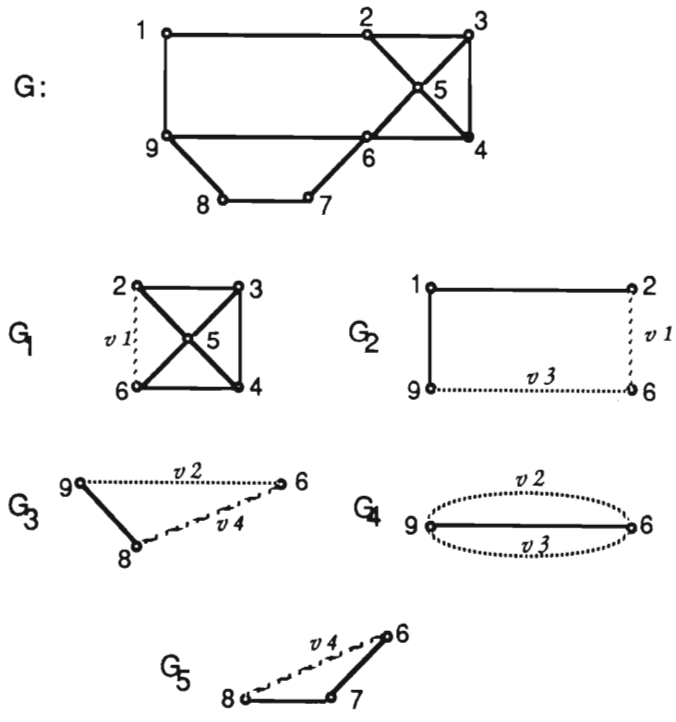


Figure 3.12 - A graph G and the split components G_1, G_2, ..., G_5 of G.

From Figure 3.12, we observe that we may obtain the triconnected components of G by merging the split components G_3 and G_5 together.

The triconnectivity algorithm is, as was already mentioned, based on the ideas used in Hopcroft and Tarjan's planarity testing algorithm of Section 2.2. The following lemma forms the basis for the triconnectivity algorithm.

Lemma 3.7: Let G be a graph, and let C be a cycle of G. Let the fragments of G with respect to C be $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_k$. Let {u, v} be a separation pair. Then the following hold.

(i) Either both u and v lie on C, or both belong to some Fragment $\mathcal{F}_i$.

(ii) Suppose u and v belong to C. Let P_1 and P_2 be the two paths on C which connect u and v. Then either

(a) Some fragment $\mathcal{F}_i$ with at least two edges has only u and v in common with C, and some vertex v_i does not lie in $\mathcal{F}_i$, or

(b) If P_1 and P_2 each contain a vertex distinct from u and v, then no fragment contains both a vertex $x \neq u, v$ which belongs to P_1 and a vertex $w \neq u, v$ which belongs to P_2.

Proof: Part (i) is easy to see. If this were not true, then suppose vertex u was on C, but v was in some fragment $\mathcal{F}_i$ to which u did not belong. Then, every component of $\mathcal{F}_i$ - {v} is still connected to C via a path which does not contain u, and C - {u} is a path. Thus, the graph G - {a, b} is still connected, which produces a contradiction.

Part(ii) is more complicated. Note that if {u, v} is a separation pair such that u and v both belong to C, then we must have that either {u, v} separate some fragment $\mathcal{F}_i$ from the rest of G, or {u, v} separates some of the fragments and part of the cycle from the rest of G. The former case satisfies condition (a), whilst it is easy to see that the latter case satisfies condition (b). ❑

Lemma 3.7 is an important lemma. Firstly, the lemma immediately suggests a recursive fragment exploration algorithm - i.e. a Depth-First Search (DFS). Secondly, it classifies the separation pairs on the cycle C into two categories. The categories are called Type 1 and Type 2. Type 1 we associate with condition (a) and Type 2 we associate with condition (b). Type 1 checking is intuitively clearer and would suggest a simpler check than that for Type 2. Recall from Section 2.2, the structure of the recursive calls for the planarity testing algorithm and the spine cycle $SC(\mathcal{F})$ of a fragment $\mathcal{F}$. The separation pairs that are not on C will be found in the recursive part of the algorithm, where we consider a new cycle, the spine cycle $SC(\mathcal{F})$ and the associated fragments.

In a manner similar to the one employed in Section 3.2, we shall make the extraction of the split components of G efficient by a judicious choice of edge orderings, and this time also by the choice of the DFS indices of the vertices. We assume, as usual, that G is 2-connected. The first part of the algorithm is a preprocessing stage, which consists of two DFS's. The first DFS computes, as well as the DFS tree, for every $v \in V(G)$ several other values. The first two values are the lowpoint values, $L_1(v)$ and $L_2(v)$. As usual, we generate, for every edge $v \xrightarrow{e} u$, a weighting $\phi(e)$ as follows :

$$\phi(e) = \begin{cases} 2.dfi(u) & \text{if } v \xrightarrow{e} u \text{ is a back edge} \\ 2.L_1(u) & \text{if } v \xrightarrow{e} u \text{ is a tree edge and } L_2(u) \geq dfi(v) \\ 2.L_1(u) + 1 & \text{if } v \xrightarrow{e} u \text{ is a tree edge and } L_2(u) < dfi(v) \end{cases}$$

Now we order the edges in the adjacency lists of each vertex in non-decreasing order of $\phi(e)$. The other value we compute, for every $v \in V(G)$, is the number of vertices in the subtree rooted at v, and is denoted by NDes(v).

The second DFS operates on the same DFS tree T as was generated by the first DFS. Also, we use the same adjacency lists which were sorted

according to increasing ø(e). We say a vertex v is *examined last* when we backtrack from it to Parent(v). In the case of Root(T), Root(T) is examined last when we finish the DFS. We reindex the vertices of T during the second DFS. The reindexing operation assigns the DFS indices from p to 1, in order, to the vertices as they are examined last.

Consider as an example the same graph shown in Figure 2.5 of Section 2.2. Figure 3.13, below, reproduces Figure 2.5.

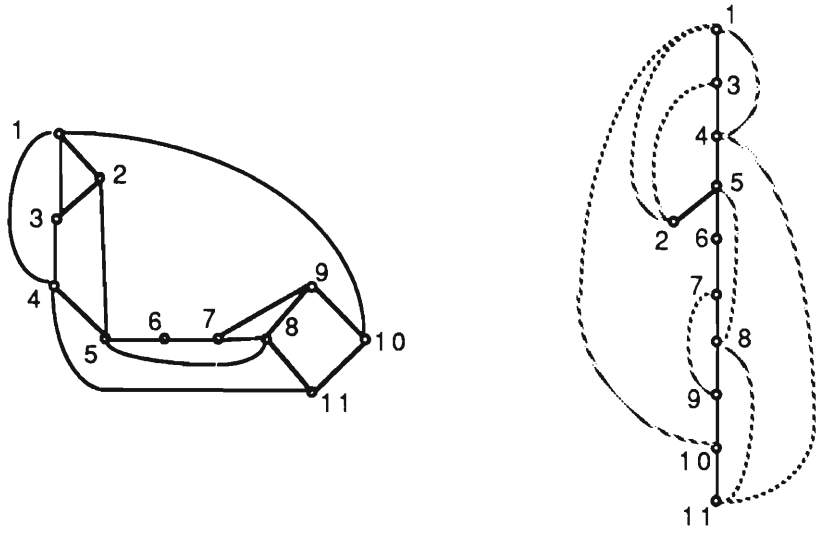


Figure 3.13 : An example graph G and its associated DFS tree (back edges are drawn as dashed lines)

From Figure 3.13, we may determine, for each vertex v, the DFS indices. Given the DFS tree, the second DFS may then compute the adjusted DFS values. Table 3.1, below, lists the relevant values.

Vertex v	dfi(v)	new dfi(v)
1	1	1
2	11	5
3	2	2
4	3	3
5	4	4
6	5	6
7	6	7
8	7	8
9	8	9
10	9	10
11	10	11

Table 3.1 - Vertices, DFS indices and Adjusted DFS indices

During the second DFS, we also, for every vertex v, calculate the vertex w with highest DFS index incident (via a back edge) to v. This value we denote by High(v) = dfi(w). Note that if no vertex w is incident along a back edge with v, then High(v) = 0.

Lastly, as a final preprocessing step, we calculate, for each vertex $v \in V(G)$, the first vertex which appears in the adjacency list of v, denoted Adj_1(v), and the degree of v, denoted Deg(v).

Once the separation pairs have been determined, finding the split components is considerably simplified. As we shall show, after preprocessing, the DFS gives a simple criterion for identifying the separation pairs of a graph G. Then, again by the preprocessing and the new numbering scheme, the determination of the split components is relatively easy. Algorithm 3.8 gives an outline of the preprocessing DFS calls and the calculation of the various values for the vertices described above.

Algorithm 3.8: Preprocess (G)

```
DFS              { DFS tree T is built and find L1(v), L2(v) and NDes (v) }
Sort the edges according to increasing ø(e).
2nd_DFS          { Renumber vertices from p to 1 in order, examined last }
For every v ∈ V(G)
    calculate Adj_1(v) and Deg(v)
```

end

The extension of the DFS to compute the second lowpoint values, and to subsequently sort the edges, has already been discussed in Section 2.2. The complexity of the algorithm remains O(p). The second DFS requires more detail. Firstly, we need to adjust the lowpoint values for the new vertex numbering. Secondly, a problem with the second DFS is that we need to number the vertices in the order in which they are visited last. But the correct calculation of the highpoint High(v) requires that we have the correct DFS indices the first time we visit a vertex. We can accomplish the calculation of the correct new DFS indices when we first visit a vertex by using the number of descendants, NDes, already calculated. Algorithm 3.9, below, gives a description of the second DFS.

Algorithm 3.9: Second_DFS (T)

```
{ We reindex the DFS indices;
      Note that Old_Number is an array [1..p],
      such that position i in Old_Number stores
      the vertex v that had
            dfi(v) = i before the reindex }

procedure Recurse (Current)

    Old_Number(dfi(Current)) = Current     { save old reference }
                                           { and compute new reference }
    dfi(Current) = Last_Assigned - NDes(Current) + 1
    For each edge e incident with Current do
        if e = Current → w is a tree edge
            then
                Recurse (w)
                Last_Assigned = Last_Assigned - 1
            else
                if High(w) = 0
                    then High(w) = dfi(Current)
end procedure
```

```
        { do initialisations first }
For each v ∈ V(T) do
    High(v) = 0
Current = Root(T)
Last_Assigned = p
        { reindex the DFS tree }
Recurse (Current)
        { and recompute the lowpoints }
For each v ∈ V(T) do
    L1(v) = dfi(Old_Number(L1(v))
    L2(v) = dfi(Old_Number(L2(v))
```

end

Lemma 3.8: Algorithm 3.9 correctly calculates new DFS indices for the DFS tree T, where vertices are numbered from p to 1, in the order that they are examined last, as well as highpoints High(v). The complexity of Algorithm 3.9 is O(p)

Proof: We note that initially Last_Assigned is set to p. Also, the variable is decremented every time we reach a new vertex. Thus, when we reach a new vertex, the value of Last_Assigned is equal to the actual new DFS numbering index we want to assign, plus the number of vertices to be assigned DFS indices before we visit the vertex for the last time. But the number of vertices to be assigned DFS indices before we visit the vertex for the last time is merely the number of descendants of that vertex. Hence, the numbering formula of Last_Assigned - NDes(v) + 1 is correct. Trivially then, since the numbering of the DFS indices is correct and the edges are ordered, by the definition of High(v), High(v) is also calculated correctly.

That Algorithm 3.9 has complexity of O(p) follows directly from the fact that Algorithm 3.9 is a DFS. ❑

We require one more definition before we can start discussing the properties of the new numbering and the various new values calculated, and how they are used to find the split components. If $v \xrightarrow{e} w$ is a tree edge, and e is the first edge in A(v), then we say that w is the *first child* of v. If $P : v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$ is a path of tree edges, and if v_i is a first child of v_{i-1} $(1 \leq i \leq n)$, then we say that v_n is a *first descendant* of v_1.

Lemma 3.9: Let $v \xrightarrow{e_1} w$ and $v \xrightarrow{e_2} u$ be two tree edges, with e_1 appearing before e_2 in A(v). Then, dfi(v) < dfi(u) < dfi(w).
Proof: Since the edge e_1 appears before the edge e_2 in A(v), w is examined last before either u or v. Also, v is examined last after u. Therefore, by the new numbering scheme, the lemma is proved. ❑

We say that an adjacency matrix that represents a 2-connected graph G has an *acceptable DFS numbering scheme* if, for every vertex $v \in V(G)$, whenever $v \rightarrow u$ is a tree edge, then dfi(v) < dfi(u), and the edges e of A(v) are sorted according to increasing ø(e).

Lemma 3.9: Suppose $L_1(v)$ and $L_2(v)$ are defined relative to some acceptable numbering scheme. Then, $L_1(v)$ and $L_2(v)$ identify unique vertices, independent of any acceptable DFS numbering scheme.
Proof: Suppose dfi(w) = $L_1(v)$. Then, since G is 2-connected, we note that w is an ancestor of v. Furthermore, since the order of the ancestors of v correspond to the order of their numbers, in any acceptable numbering scheme, w is the unique vertex for which dfi(w) = $L_1(v)$ (i.e. vertex w corresponds to the first ancestor of v reachable from v, by a path of tree edges followed by a back edge, on the path from the root of T to v). The proof for $L_2(v)$ is the same. ❑

Lemma 3.11: The new numbering scheme, calculated in Algorithm 3.9, is an acceptable numbering scheme.
Proof: If $v \rightarrow u$ is a tree edge, then v is visited last after u, thus dfi(v) < dfi(u). Then, from Lemma 3.10 the proof follows, since the edges are still sorted according to increasing ø(e). ❑

Thus, the new numbering scheme calculated in Algorithm 3.9 is acceptable, in the sense that the properties of the tree and back edges of T remain the same. Now consider the adjustment of lowpoint values performed in Algorithm 3.9. Firstly, we note the vertices which the DFS indices correspond to. Then, we change the lowpoint values to the new DFS indices of the vertices which the lowpoint values corresponded to

previously. Thus, by Lemma 3.10, we conclude that the adjustment of the lowpoints in Algorithm 3.9 is correct.

The next lemma indicates the usefulness of the new numbering scheme. Let $v \in V(G)$ be a vertex. We define $\mathcal{D}(v)$ to be the vertices in the subtree rooted at v.

Lemma 3.12: If $v \in V(G)$ is a vertex, then

$$\mathcal{D}(v) = \{x \mid dfi(v) \leq dfi(x) \leq dfi(v) + NDes(v)\}$$

and if w is a first descendant of v, then

$$\mathcal{D}(v) - \mathcal{D}(w) = \{x \mid dfi(v) \leq dfi(x) < dfi(w)\}$$

Proof: The proof follows easily from the definition of the new numbering scheme. The vertices in the subtree rooted at v will receive the DFS indices from dfi(v) to dfi(v) + NDes(v), since we never backtrack from a vertex u in the DFS until the entire subtree rooted at u has been explored. But then, by a similar argument, the subtree rooted at w will receive the DFS indices $dfi(w) \leq dfi(x) \leq dfi(w) + NDes(w)$. Since w is a first descendant, the subtree rooted at w will be the first part of the subtree rooted at v to be explored in the DFS. The lemma then follows. ❑

We shall show later that the separation pairs consist, in certain cases, of ancestors and first descendants. Thus, once the separation pair has been determined, Lemma 3.12 provides an easy criteria to identify the vertices of the split component corresponding to the subtree rooted at w.

Lemma 3.13: Let {a, b} be a separation pair in G with dfi(a) < dfi(b). Then there exists a (unique) path P of tree edges, where $P : a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n = b$.

Proof: Since dfi(a) < dfi(b), the vertex a cannot be a descendant of b. Suppose that b is not a descendant of a. Now, neither a nor b can be the root of T. Since G is 2-connected, we have, for every tree edge $e_1 = a \rightarrow v$, that $L_1(v) < dfi(a)$, and for every tree edge $e_2 = b \rightarrow v$, $L_1(v) < dfi(b)$. Let P_a and P_b be the paths from the root to a and b respectively in the DFS tree. Since a is not an ancestor of b, a is not in P_b. Now, note that every descendant of b is connected to some vertex in P_b different from b, and

that every descendant of a is connected to some vertex in P_a different from a. Thus, the removal of a and b does not disconnect G. ❑

Lemma 3.13 indicates that we may obtain the separation pairs of a graph G by looking at ancestor, descendant pairs in G. Theorem 3.2, below, gives a clear result of how to detect the separation pairs of G.

Theorem 3.2: Suppose dfi(a) < dfi(b). Then, {a, b} is a separation pair of G if and only if a is an ancestor of b, and if one of the following two cases holds.

(i) There are distinct vertices $r \neq a, b$ and $s \neq a, b$, such that there is a tree edge $b \rightarrow r$, $L_1(r) = dfi(a)$, $L_2(r) \geq dfi(b)$ and s is not a descendant of r.

(ii) There is a vertex $r \neq b$, such that $a \rightarrow r$ is a tree edge, and b is a first descendant of r, $dfi(a) \neq 1$, and the following conditions about back edges are true. If $x \rightarrow y$ is a back edge with $dfi(r) \leq dfi(x) < dfi(b)$, then $dfi(a) \leq dfi(y)$. Also, if $x \rightarrow y$ with $dfi(a) < dfi(y) < dfi(b)$ and $x \in \mathcal{D}(w)$, where $b \rightarrow w$ is a tree edge, then $L_1(w) \geq dfi(a)$.

Proof: We prove the converse first. Let E_i $(1 \leq i \leq k)$ be the separation classes of G with respect to {a, b}. Suppose that a pair {a, b} satisfies condition (i) of Theorem 3.2. Then the tree edge $b \rightarrow r$ belongs to some separation class E_1. Since $L_1(r) = dfi(a)$ and $L_2(r) \geq dfi(b)$, every edge in E_1 with an endpoint in $\mathcal{D}(r)$ must have the other endpoint in $\mathcal{D}(r) \cup \{a, b\}$. Note that every edge in E_1 has an endpoint in $\mathcal{D}(r)$, and that every edge with an endpoint in $\mathcal{D}(r)$ belongs to E_1. Hence, E_1 consists of all edges with an endpoint in $\mathcal{D}(r)$. Since there is another vertex $s \neq a, b$ which is not a descendant of r, there is another, non-trivial, separation class E_2. Therefore, there are two non-empty separation classes with respect to {a, b} and so {a, b} is a separation pair.

If the pair {a, b} satisfy condition (ii), then the edge $a \rightarrow r$ belongs to some separation class E_1 say. Consider the set $S = \mathcal{D}(r) - \mathcal{D}(b)$. Because every back edge $x \rightarrow y$ with $dfi(r) \leq dfi(x) < dfi(b)$ is such that $dfi(a) \leq dfi(y)$, all edges incident from vertices in S belong to E_1. Also, if $x \rightarrow y$ is a back edge such that $dfi(a) < dfi(y) < dfi(b)$, x is a descendant of w, where $b \rightarrow w$ is a

tree edge, then $L_1(w) \geq dfi(a)$, and thus all edges incident to vertices in S belong to E_1.
We show now that the root cannot belong to E_1. Suppose it does. Then this would either imply a back edge from a vertex y with $dfi(a) < dfi(y) < dfi(b)$ to an ancestor of a, or a back edge from a descendant of a vertex w, where $b \rightarrow w$ is a tree edge, to an ancestor of a, with another back edge from $\mathcal{D}(w)$ to a vertex y with $dfi(a) < dfi(y) < dfi(b)$. In either case we have a contradiction of the hypothesis of the theorem. Thus, the root belongs to another separation class E_2, and since G is 2-connected E_2 is non-trivial. Thus, {a, b} is a separation pair. That a is an ancestor of b if {a, b} is a separation pair follows from Lemma 3.13.

The direct part of Theorem 3.2 is harder to prove. Suppose that {a, b} is a separation pair with $dfi(a) < dfi(b)$. Then, let $E_1, E_2, ..., E_k$ be the separation classes, $k \geq 2$. Suppose $e = a \rightarrow v$ is a tree edge, where $b \in \mathcal{D}(v)$. Let $S = \mathcal{D}(v) - \mathcal{D}(b)$. Let $X = V(G) - \mathcal{D}(a)$. Either X or S may be empty. Suppose that $E(< S >) \subseteq E_1$ and that $E(< X >) \subseteq E_2$. We now describe what the separation classes of G with respect to {a, b} will be.
For a vertex $v \in V(G)$, define $E(\mathcal{D}(v))$ to be the set of edges with an endpoint in $\mathcal{D}(v)$. Suppose there exists a child $a_i \neq v$ of a. Then, since $L_1(a_i) < a$, $E(\mathcal{D}(a_i)) \subseteq E_2$. Let $Y = X \cup \{\mathcal{D}(a_i) \mid 1 \leq i \leq m\}$, where $a_1, a_2, ..., a_m$ are the children of a different from v. Let $b_1, b_2, ..., b_n$ be the children of b in the order in which they occur in A(b). The separation classes must be unions of the sets $E(< S >)$, $E(< Y >)$, $\{(a, b)\}$, $E(\mathcal{D}(b_1))$, $E(\mathcal{D}(b_2))$, ..., $E(\mathcal{D}(b_n))$. To prove that {a, b} satisfies condition (i) or condition (ii), we have two cases.

Case 1: Suppose $E(\mathcal{D}(b_i)) = E_j$ for some i, j. Then we must have that $L_1(b_i) = dfi(a)$ and that $L_2(b_i) \geq dfi(b)$ (or else $E(\mathcal{D}(b_i))$ would not be the separation class E_j). Since {a, b} is a separation pair, we have at least one other separation class different from E_j and $\{(a, b)\}$. Thus, there is a vertex s such that $s \neq a, b$ and $s \notin \mathcal{D}(b_i)$. Thus, {a, b} satisfies condition (i), with $r = b_i$.
Case 2: Suppose now that no $E(\mathcal{D}(b_i))$ is a separation class. Let $i_0 = \min \{ i \mid L_1(b_i) \geq dfi(a) \}$. If $i \geq i_0$, then, since G is 2-connected, $L_1(b_i) < dfi(b)$ and the separation classes are $E_1 = E(< S >) \cup (E(\mathcal{D}(b_i) \mid i \geq i_0))$, $E_2 = E(< Y >)$

$\cup$ ($E(\mathcal{D}(b_i) \mid i<i_0)$) and, if edge $a \rightarrow b$ exists, $E_3 = \{(a, b)\}$. If $e = a \rightarrow b$ is a tree edge, then condition (i) would have been fulfilled. Therefore, $v \neq b$. If $x \rightarrow y$ is a back edge with $dfi(v) \leq dfi(x) < dfi(b)$, then $dfi(a) \leq dfi(y)$, or else E_1 and E_2 are not distinct and {a, b} is not a separation pair. Similarly, every back edge $x \rightarrow y$ with $dfi(a) < dfi(y) < dfi(b)$ and $b \rightarrow b_i$, where x is a descendant of b_i has $L_1(b_i) \geq dfi(a)$ and $i \geq i_0$.

To show condition (ii) is satisfied, we have to show that b is a first descendant of v. Suppose b is not a first descendant of v. Then, we know by the ordering of the adjacency lists that there exists a back edge $x \rightarrow y$ with $x \in \mathcal{D}(v)$ and $dfi(y) < dfi(a)$, such that x is a first descendant of v. If b was not a first descendant, then $x \in S$ and E_1 and E_2 would not be distinct. Thus, b is a first descendant of v and condition (ii) holds. ❑

Theorem 3.2 thus provides two easily applied conditions to find the separation pairs {a, b}. The first condition is equivalent to the Type 1 condition of Lemma 3.7, whereas the second condition is equivalent to Type 2 condition of Lemma 3.7.

Recall that the first stage of the algorithm for finding separation pairs is a preprocessing step. The next stage of the algorithm applies another DFS to the graph, and checks for pairs of vertices satisfying the above conditions. Type 1 separation pairs are easy to find. At each stage in the DFS, we merely check the vertex v and its lowpoint $L_1(v)$ to see if they satisfy condition (i) in Theorem 3.2. The second condition is more difficult to check. To do this we keep a list of candidate pairs, which we may check during the DFS for satisfying one of the two criteria of condition (ii) in Theorem 3.2.

Notice how the new numbering scheme simplifies the extraction of the split components from G. Let {a, b} be a Type 2 pair satisfying $a \rightarrow r$ and $b \in \mathcal{D}(r)$, and $i_0 = \min \{ i \mid L_1(b_i) \geq dfi(a) \}$, with b_i $(1 \leq i \leq n)$ the children of b. A split component of G with respect to {a, b} is given by $E(<\{x \mid dfi(r) \leq dfi(x) < dfi(b_{i_0}) + NDes(b_{i_0}) \} - \{b\}>)$. Thus, if we keep a stack of edges which we have scanned, then we may remove from the stack the edges which satisfy the above criterion.

We define Type 2a pairs to be Type 2 separation pairs {a, b} for which there exists a path $a \rightarrow v \rightarrow b$, where v has degree 2. Type 2b pairs are all other Type 2 separation pairs. We then separate the separation pair finding into three cases. Type 1 pairs and Type 2a and Type 2b pairs. As mentioned earlier, Type 1 pairs are checked for easily by merely looking at the vertex w with dfi(w) = lowpoint $L_1(v)$ and checking that w and v satisfy condition (i) of Theorem 3.2. Type 2a separation pairs are also easy to find, since there are no back edges incident to or from v. Thus, $V(E_1)$ = {a, v, b}. Type 2b separation pairs are the most difficult pairs to detect.

During the running of the algorithm, we keep a stack of edges, which we add to the stack when we backtrack along an edge. This stack we call *Edges*. As already discussed, when we discover a split component, we merely remove all edges corresponding to that split component from the stack. We then add the virtual edge both to the split component and to the edge stack. Care must be exercised when splitting the graph, since the degrees of vertices may change, as may parent values. To check for separation pairs of Type 2b, we keep a stack of potential Type 2b separation pairs. The stack we call *Triples*, and each element is a triple of the form (a, b, h), where {a, b} is the candidate separation pair, and h is the highest DFS index in the corresponding split component. As with the planarity testing algorithm of Section 2.2, we use the concept of a spine cycle by Mehlhorn [Meh84]. The elements of Triples are arranged in nested order. If v is the current vertex in the DFS, and (a_1, b_1, h_1), (a_2, b_2, h_2), ..., (a_k, b_k, h_k) are the entries on the stack (so (a_1, b_1, h_1) is the top-of-stack element), then $dfi(a_k) \leq dfi(a_{k-1}) \leq \ldots \leq dfi(a_1) \leq dfi(v) \leq dfi(b_1) \leq dfi(b_2) \leq \ldots \leq dfi(b_k)$. Furthermore, a_i and b_i are both on the spine cycle C. Triples is updated in the following manner.

1. Every time we start a new path (i.e. we have stopped from backtracking up the tree, and have found a new edge), then we may delete certain triples.
   Let the new path followed have first vertex v and last vertex w, i.e. w is an ancestor of v. We may delete all triples (a, b, h) on top of the stack with dfi(a) > dfi(w). These triples are invalid because

there is a back edge from a vertex x such that dfi(w) < dfi(x) ≤ dfi(v) to an ancestor of a, namely w.

We then insert a new guess triple, depending on whether any triples were deleted. If the new edge e = v → u is not a back edge, then we let x = dfi(u) + NDes(u) - 1, otherwise let x = dfi(v). Let the triples deleted from Triples be (a_j, b_j, h_j), $1 \le j \le k$. Let $y = \max\{ h_j \mid (a_j, b_j, h_j)\ (1 \le j \le k)\}$, otherwise let y = 0. Now, if a triple was deleted, then we make a new guess, namely the triple $(w, b_k, \max(x, y))$. Here we are merging all triples which had back edges incident to a vertex x with dfi(w) < dfi(x) < dfi(v). If no triple was deleted then the new guess is (w, v, x).

2. When backtracking along a tree edge $v_i \rightarrow v_{i+1}$, we delete all entries (a, b, h) on the top of Triples, such that $High(v_i) > h$ and $dfi(b) > dfi(v_i)$. Triples of this form are invalid because there are back edges from elements corresponding to children b_i of b with $i < i_0$ incident to the vertex v_i. By Theorem 3.2, this disqualifies the triple immediately. Note that Hopcroft and Tarjan [HT73b] failed to observe the second part of the above condition, i.e. the check $dfi(b) > dfi(v_i)$.

Now, given the triple stack it is easy to check for Type 2b separation pairs. In general, when we backtrack along a tree edge $v_i \rightarrow v_{i+1}$ we examine the stack Triples, and the top triple (a_1, b_1, h_1).

If $dfi(v_i) \neq 1$, $a_1 = v_i$ and $a_1 \neq Parent(b_1)$, then (a_1, b_1) is a Type 2b separation pair (we justify this in Theorem 3.3).

If $Deg(v_{i+1}) = 2$ and v_{i+1} has a child w (i.e. $NDes(v_{i+1}) > 1$), then $\{v_i, w\}$ form a Type 2a separation pair.

We test for condition (i) from Theorem 3.2 at vertex v_i. If it succeeds, then, if w is the vertex corresponding to $L_1(v_i)$, then $\{v_i, w\}$ is a Type 1 separation pair.

The above tests need to consider the case of multiple edges. If {a, b} is a separation pair, then an edge may already exist between vertex a and vertex b. Thus, we must test for this case before we add the virtual edge onto the stack. The multiple edge case is handled easily because of the

second lowpoint values as we now see. Suppose that v is a vertex, and v_1, v_2, ..., v_k are children of v as they appear in A(v), such that $L_1(v_i) = dfi(u)$, $(1 \leq i \leq k)$. Suppose that there is also a back edge $v \rightarrow u$. If A(v) is sorted, then there is some i_0 such that for all $i > i_0$, $L_2(v_i) < dfi(v)$, and for all $i \leq i_0$, $L_2(v_i) \geq dfi(v)$. By the ordering function $\phi(e)$, the back edge $v \rightarrow u$ will appear with all edges $v \rightarrow v_i$ $(i \leq i_0)$, and before the edges $v \rightarrow v_i$ $(i > i_0)$. Now, if $i \leq i_0$, then we have a Type 1 separation pair, and by examining the top element of the stack of edges, we may check if there are already edges of the form $v \rightarrow u$. In this case we have a multiple edge. When we add the back edge $v \rightarrow u$ to the edges stack, as we shall show, we may use the field Adj_1(v) to check that a multiple edge case does exist (i.e. a virtual edge $v \rightarrow u$ has been added to the edges stack). If the edges were ordered by our ordering function $\phi(e)$, then we would be unable to discern if we have encountered a multiple edge case.

Lastly, the recursive exploration of fragments must be considered. Starting a new path which starts with a vertex on the cycle C, implies that we are checking a fragment for separation pairs. When we have completed all the checks as described above, we place an end-of-stack marker on the stack Triples, recursively test the fragment for separation pairs, and then when we backtrack along the first vertex of the new path, we delete all triples from the stack Triples until we reach the end-of-stack marker, which we then remove.

We are now ready to present the algorithm to find the split components of a 2-connected graph G.

Algorithm 3.10: Split_Components (G)

```
{ Find the separation pairs and split components of G }

procedure Recursive_Search (Current)

    For each edge e ∈ A(Current) do

        Let e be of the form e = Current → w
        if dfi(w) > dfi(Current)
            then                                    { tree edge }
                if e is the first edge on a new path
                    then
```

```
                        Update Triples stack          { point 1, above }
                        Add end-of-stack marker to Triples
                    Recursive_Search (w)              { proceed down the path }
                    Add e to Edges stack
                    Check for Type 2a and Type 2b     { check for pairs }
                    Check for Type 1
                    Update Triples stack                  { point 2, above }
                else                                  { back edge }
                    if e was first and last edge of a path
                        then
                            Update Triples stack      { point 1, above }
                    Use Adj_1(v) to check for multiple edges
                     Add e to Edges stack
end procedure

DFS                                   { preprocessing }
Order_Edges
Second_DFS
For every v ∈ V(G)
    calculate Adj_1(v) and Deg(v)
Comp_No = 0                           { the virtual label we are using }
Edge stack = ∅ and Triple stack = ∅       { initialisation }
Recursive_Search (Root)               { perform the DFS }
Comp_No = Comp_No + 1                 { all other edges in component }
Add all edges on Edge stack to last split component
```

end

To assign a label to a virtual edge, we use the variable Comp_no. Also, note that at the end of the algorithm we add all remaining edges on the stack Edges to a last split component. Consider having removed all triconnected components from the graph. The remaining graph is also triconnected, and thus is also a triconnected component. The code description for the updating of the stack Triples has already been covered. The checking for separation pairs will be presented next. Algorithm 3.12 presents the code for checking for Type 2 separation pairs. Note that if an edge of stack Edges does not have a virtual label, then we assign it the virtual label of zero. This is done to generalise some of the code.

Algorithm 3.11: Type_2_Separation_Pairs (Current)
{ we check for Type 2 separation pairs }

```
Multiple = false                  { no multiple edge condition }
while (Current ≠ 1)           { w is vertex from Algorithm 3.10 }
        and (((Deg(w) = 2) and (dfi(Adj_1(w)) > dfi(w))
            or
              ((a, b, h) on Triples satisfies (Current = a)) do
```

```
                if (Current = a) and (Parent(b) = a)
                    then                        { disqualified - Theorem 3.2 }
                        Delete triple from Triples stack
                    else
                        if (Deg(w) = 2) and (dfi(Adj_1(w)) > dfi(w))
                            then                { easy case vertex deg 2, Type 2a}
                                Comp_No = Comp_No + 1
                                Let top two of Edge stack be (Current, w) and (w, x)
                                Let b = x       { sep. pair is {Current, b} }
                                Add top two edges to new component
                                Add virtual edge (Current, x, Comp_No)
                                if (y, z) on Edge stack has (y, z) = (Current, x)
                                    then        { multiple edge }
                                        Multiple = true
                                        Save_No = virtual number
                            else if Current = a
                                then            { success - Type 2b }
                                    Delete triple (a, b, h) from Triples
                                    Comp_No = Comp_No + 1
                                    while (x, y) on Edge stack has
                                        (dfi(a) ≤ dfi(x) ≤ dfi(h))
                                            and (dfi(a) ≤ dfi(y) ≤ dfi(h)) do
                                        if (x, y) = (a, b)
                                            then    { multiple edge }
                                                Multiple = true
                                                Save_no = virtual number
                                            else
                                                Delete (x, y) from Edge stack
                                                Add(x, y) to component
                                                Decrement Deg(x) and Deg(y)
                                    Add (a, b, Comp_No) to component
                        if Multiple
                            then        { triple bond component }
                                Multiple = false
                                Comp_No = Comp_No + 1
                                Add (b, a, Comp_No - 1) to component
                                Add (b, a, Comp_No) to component
                                Add (b, a, Save_No) to component
                                Decrement Deg(b) and Deg(a)
                        Add (Current, b, Comp_No) to Edge stack          { add virtual
edge }
                        Increment Deg(Current) and Deg(b)
                        if dfi(b) ≤ dfi(Adj_1(Current)) + NDes(Adj_1(Current)
                            then Adj_1(Current) = b          { adjust Adj_1(a) }
                        w = b                                { next edge now b }
end
```

Note how the degrees of vertices are also adjusted. This is to facilitate the discovery of Type 2a separation pairs in split graphs.

Algorithm 3.12: Type_1_Separation_Pairs (Current)

{ we check for Type 1 separation pairs }

```
if (L2(w) ≥ Current) and          { possible pair (a, b) = (w, L1(w)) }
    ((L1(w) ≠ 1)                              { root is above a }
    or (dfi(Parent(Current)) ≠ 1)  { or ancestors of w exist }
    or (dfi(w) > 3))                          { or Current has other descend }
  then
    Comp_No = Comp_No + 1
    while (x, y) on Edge stack has
        (dfi(w) ≤ dfi(x) ≤ dfi(w) + NDes(w))
      and (dfi(w) ≤ dfi(y) ≤ dfi(w) + NDes(w)) do   { add all edges }
        Delete (x, y) from Edge stack
        Add(x, y) to component
        Decrement Deg(x) and Deg(y)

    Add (w, L1(w), Comp_No) to component        { virtual edge }
    if Adj_1(Current) = w                       { adjust Adj_1 }
        then Adj_1(Current) = L1(w)

    if (x, y) on Edge stack has (x, y) = (w, L1(w))  { multiple edge }
        then
            if the edge on Edge stack has component no Save_no
            Comp_No = Comp_No + 1
            Add (w, L1(w), Comp_No - 1) to component
            Add (w, L1(w), Comp_No) to component
            Add (w, L1(w), Save_No) to component
            Decrement Deg(w) and Deg(L1(w))
    if L1(w) ≠ Parent(Current)
        then                          { add virtual edge }
            Add (Current, L1(w), Comp_No) to Edge stack
            Increment Deg(Current) and Deg(L1(w))
        else                          { multiple edge }
            Comp_No = Comp_No + 1
            Add (Current, L1(w), Comp_No - 1) to component
            Add (Current, L1(w), Comp_No) to component
            Add (Current, L1(w), Save_No) to component
            Mark tree edge (w, L1(w)) on Edge stack as Comp_no
```

end

Theorem 3.3: Algorithm 3.10 correctly divides a graph G into split components.

Proof: The tests for multiple edges, and for Type 1 and Type 2a separation pairs are straightforward. They will discover a separation pair if one exists, and will not report one if it does not exist. Thus, we focus on the

maintenance of the stack Triples, and the testing for separation pairs Type 2b using the stack Triples.

We consider the operation of Algorithm 3.10 on a graph G with no Type 1 and Type 2a separation pairs. If (a_1, b_1, h_1), (a_2, b_2, h_2), ..., (a_k, b_k, h_k) are the entries on Triples, then $dfi(a_k) \leq dfi(a_{k-1}) \leq ... \leq dfi(a_1) \leq dfi(v) \leq dfi(b_1) \leq dfi(b_2) \leq ... \leq dfi(b_k)$. This can be easily seen by observing when the stack Triples changes. Also, we note that each a_i and b_i lie on the current cycle C.

Suppose now that a triple (a, b, h) on stack Triples was discovered by the program as a separation pair, during the Type 2b testing phase. Therefore, the following conditions have succeeded, (Current = a), (Parent(b) $\neq$ a) and (dfi(Current)$\neq$ 1). We let $a \rightarrow r$ be such that $b \in \mathcal{D}(r)$, and we know that Adj_1(a) = r. Thus a, r, and b lie on a commonly generated path, and b is a first descendant of a. If b was not on the commonly generated path, then the triple (a, b, h) would not have been added to the Triples stack, since we only add triples when we reach the end of each path which we generate. Now, we look at the back edges incident from and to vertices on the path from a to b.

If there is a back edge $x \rightarrow y$ with $dfi(r) \leq dfi(x) < dfi(b)$ and $dfi(a) > dfi(y)$, then the triple would have been deleted at the start of a new path check (point 1 in the discussion preceding Algorithm 3.10). Similarly, if there is a back edge $x \rightarrow y$ with $dfi(r) \leq dfi(y) < dfi(b)$, then suppose that $b \rightarrow w$ and $x \in \mathcal{D}(w)$. We have that if $L_1(w) < dfi(a)$, then the triple would have been deleted in the test of High(y). Thus, by Theorem 3.2, {a, b} is indeed a separation pair.

Now, let {a, b} be a Type 2b separation pair. We will show that, given {a, b} is a separation pair, Algorithm 3.10 will find a separation pair in G. Since the split components are not unique, we are unable to guarantee that {a, b} will be found. Let the children of b be $b_1, b_2, ..., b_n$. Then, let $i_0 = \min \{ i \mid L_1(b_i) \geq dfi(a) \}$. If i_0 exists, then the triple $(L_1(b_i), b, dfi(b_i) + NDes(b_i))$ will be placed on the stack when we explore tree arc $b \rightarrow b_{i_0}$. If this triple is deleted, then it will always be replaced with a triple of the form (x, b, h), where $L_1(b_i) \geq dfi(x) \geq dfi(a)$. Eventually, such a triple will satisfy the Type 2b test.

If i_0 does not exist, then let $v_i \rightarrow v_j$ be the first edge traversed after b is reached such that dfi(a) $\leq$ dfi(v_i) and dfi(v_j) $\leq$ dfi(b). If $v_i \rightarrow v_j$ is a tree edge, then (v_i, $L_1(v_j)$, dfi(v_j) + NDes(v_j)) will be placed on Triples, possibly modified and eventually selected as a Type 2b pair, unless some other pair is selected first. If $v_i \rightarrow v_j$ is a back edge, then (v_i, v_j, dfi(v_i)) will be placed on Triples, possibly modified and eventually selected as a Type 2b pair, unless some other pair is selected first. These triples will never get deleted, because they satisfy the conditions for Theorem 3.2 condition (ii). Thus, a Type 2b triple will be discovered.
Hence, a separation pair is discovered by Algorithm 3.10 if and only if {a, b} is a separation pair. By induction on the number of edges in G, it is easy to see that since we apply the algorithm to the split graphs, Algorithm 3.10 finds the split components of G. ❑

Theorem 3.4: Algorithm 3.10 has complexity O(p).
Proof: By Lemma 3.5 the number of edges in the split graphs is bounded by 3q - 6, so the number of edges is O(q) = O(p). From Lemma 3.8 the complexity of the second DFS is O(p), and it is easy to see that the other preprocessing steps of Algorithm 3.8 are O(p). Now, turning to the procedure Recursive_Search given in Algorithm 3.10, we note that each edge of the split components is added to the stack Edges at most once and deleted once. Also, the actual DFS performed in the procedure requires O(p) steps to complete. The number of triples added to the stack Triples is bounded by the number of edges, and is thus also O(p), and each triple is modified only if it is on top of the stack. So, since the DFS is performed in O(p) time, the modification of the triples is performed in O(p) time as well. ❑

We note that Hopcroft and Tarjan provided a more general result to that which we have discussed. They allow any graphs with multiple edges, and have an extra condition for Theorem 3.2. The complexity of their algorithm is O(p + q).

Section 3.3

Karabeg
Triconnectivity Testing by PQ-Trees

In this section we discuss a linear time triconnectivity testing algorithm by Karabeg [Kar89]. The algorithm is a modified version of the planarity testing algorithm by Lempel, Even and Cederbaum [LEC67], which was presented in Chapter 2. We follow Karabeg [Kar89] for some details.

We assume that we have a 2-connected planar graph G, with a valid st-numbering. Furthermore, we restrict our discussion to graphs with minimum degree greater than 2. As we shall show, the restriction of the algorithm to graphs with minimum degree greater than 2 is not a serious restriction. To describe the algorithm, we begin by observing several relationships between PQ-trees and separation pairs. Note once again, that the virtual edges referred to in this section should not be confused with the virtual edges which were used in connection with bush forms. As before, we direct edges from vertices of lower st-number to vertices of higher st-number to obtain a digraph D. We refer to vertices by their st-numbers.

Lemma 3.14 If {a, b}, a < b, is a separation pair of G, then there exists a (directed) path P from a to b in D.

Proof : Since {a, b} is a separation pair, we have at least two non-trivial separation classes with respect to {a, b}. One class, E_1 say, will always contain the edge $1 \rightarrow p$. If E_1 only contains the edge $1 \rightarrow p$ then, since {a, b} is a separation pair, a = 1 and b = p, and the result follows. Suppose now that E_1 does not only contain the edge $1 \rightarrow p$. Consider any non-trivial separation class E_2 with respect to {a, b}, not containing the edge $1 \rightarrow p$. Take any vertex $v \in V(E_2)$. Then, by the definition of st-numbering, there is a path from 1 to v, passing through a (or else v would belong to the separation class containing the edge $1 \rightarrow p$). Similarly, there is a path from v to p, passing through b. Thus, the lemma is proved. ❑

Lemma 3.14 implies that if we discover a directed a - b path during the b-reduction of the corresponding PQ-tree T, then {a, b} is a potential separation pair. In an analogous fashion to the triconnectivity algorithm by Hopcroft and Tarjan, see Lemma 3.13, a separation pair forms an ancestor, descendant pair in terms of st-numbering.

In order to correctly identify a separation pair {a, b} in a PQ-tree T, we need to know when each node was inserted into T. We define the *label* of a P-node to be the st-number of the graph vertex that introduced this P-node into the PQ-tree. There are two cases when a P-node is inserted into a PQ-tree. After a reduction for a vertex with st-number k, we substitute the full nodes with a single P-node, having children which are leaves which represent edges directed out of k. Here the P-node is assigned label k. Also, during the applications of Templates P_2, P_3, ..., P_6, we may possibly introduce a new P-node. In this case we give the P-node the label of the P-node which was considered in the template matching. This is easy to see if we note that the new P-node contains a subset of the children of the old P-node, and thus has the same origin, in terms of graph vertices. Similarly, we define the *joint* of a Q-node to be the lowest numbered vertex in the 2-connected subgraph which the Q-node represents. In Templates P_4 and P_6, the joint of the Q-node, explicitly shown in the replacement pattern, is assigned to be the label of its P-node parent. This assignment is valid because, in all the cases, we are constraining all or part of the children of the P-node to a certain pattern. Thus, the Q-node is an extension of the P-node, and certainly, we must have that the joint equals the label of the parent. For Templates P_3 and P_5, where the Q-node replaces the P-node which matched the template, the joint of the Q-node is not yet assigned a value.

The following lemma provides a condition to identify a separation pair {a, b}, where $a < b$.

Lemma 3.15: If, in the process of the b-reduction ($a \neq 1$ and $b \neq 1$), one of the following conditions occur, then {a, b} is a separation pair.

(i) We encounter a full P-node whose label is a.

(ii) We encounter a full Q-node whose joint is b.

Proof : We show that the occurrence of a full node N, as in the statement of Lemma 3.15, determines at least two non-trivial separation classes. We split the proof into two cases, depending on whether N is a P-node or a Q-node :

If N is a full P-node whose label is a, then all of N's children were labelled full by the reduction algorithm. This implies that each child represents a separation class (see Figure 3.14a). We note that at most one of them may be trivial, thus, for a P-node with more than two children, the result is immediate. If N has exactly two children, then one may possibly be trivial. However, we observe that, since $a \neq 1$ and $b \neq 1$, we have from Lemma 3.14 that there is a non-trivial separation class which contains the edge $1 \rightarrow p$.

If N is a Q-node whose joint is labelled full, then the 2-connected component represented by the Q-node has, as vertex with greatest st-number, vertex b. That N is labelled full implies that all paths, from vertices in the 2-connected component represented by N, to vertices of higher st-number pass through b (see Figure 3.14b). Thus, we have that the 2-connected component represented by N forms one non-trivial separation class, whilst the separation class containing the edge $1 \rightarrow p$ is another. ❑

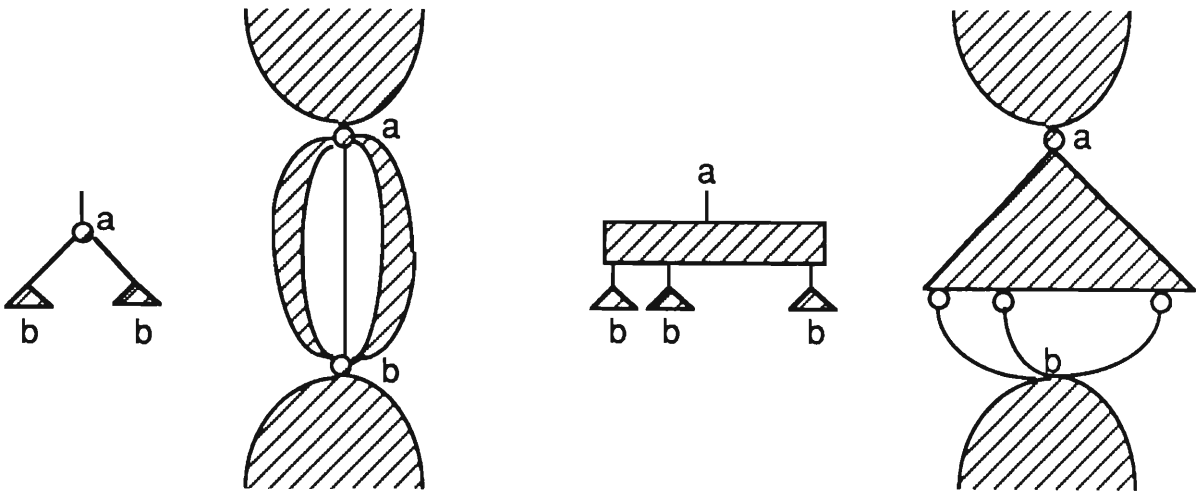


Figure 3.14 - Proof of Lemma 3.15

The converse of Lemma 3.15 is not true. However, we may complete the identification of separation pairs with the following lemmas.

Lemma 3.16: If {a, b} is a separation pair, and d is the only higher labelled vertex adjacent to b, then {a, d} is a separation pair.
Proof: See Figure 3.16. If {a, b} is a separation pair, then we may obtain the separation classes corresponding to all vertices v with $a \leq v \leq b$. We replace these separation classes with a virtual edge $e = a \rightarrow b$. Now, note that the virtual edge $e = a \rightarrow b$ together with the edge $b \rightarrow d$ form a separation class with respect to the separation pair {a, d}. This separation pair is analogous to the separation pair Type 2a of the preceding section. Note that if a = 1 and d = p, then the edge $1 \rightarrow p$ is a trivial class. However, the restriction that all vertices have degree greater than 2 guarantees the existence of at least two non-trivial separation classes with respect to the pair {1, p}. ❑

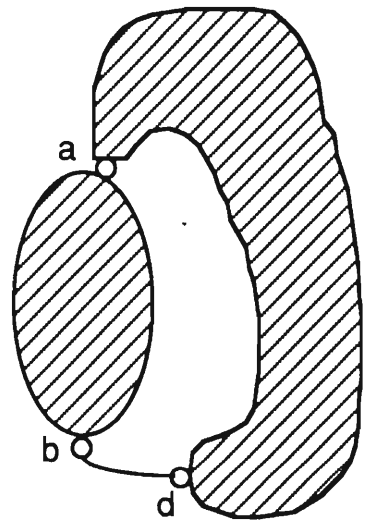


Figure 3.15 - Proof of Lemma 3.16

Note that Lemma 3.16 does not allow for repeated applications. Since we assume d has degree greater than 2, there are at least two other vertices adjacent to d.

Lemma 3.17: If, in the process of b-reduction, we encounter a full P-node (Q-node) whose label (joint) a is different from that of its P-node parent, with label c and where $\{c, b\} \neq \{1, p\}$, then {c, b} is a separation pair.

Proof: The proof is analogous to that of Lemma 3.16. See Figure 3.16. From Lemma 3.15, {a, b} is a separation pair. We replace the separation classes corresponding to vertices v with $a \leq v \leq b$, with a virtual edge $a \rightarrow b$. Now, either the pair {c, a} is a separation pair, so $c \rightarrow a$ is a virtual edge, or the only vertex of lower st-number than a, adjacent to a, is c (so $c \rightarrow a$ is a graph edge). In either case {c, b} is a separation pair, analogous to a Type 2a separation pair. ❑

Note that Lemma 3.17 is only valid if the parent is a P-node. A parent Q-node corresponds to a 2-connected component in the corresponding graph, so a is adjacent to other vertices v with $v < a$, or with $a < v < b$, but not in the separation class with respect to {a, b}. Thus, there is no virtual edge $c \rightarrow a$, and the argument used in Lemma 3.17 does not follow.

Lemma 3.17 implicitly requires that chains of P-nodes with a single child be allowed. If, after the application of Template P_4, the Q-node is the only child of the P-node, then we may not remove the P-node from the PQ-tree. If the Q-node proves to be full at a later stage in the reduction pass, for a vertex b, say, then we will have a triangle split component. This component will be represented by the vertex b, the label of the Q-node, and the label of the parent of the P-node.

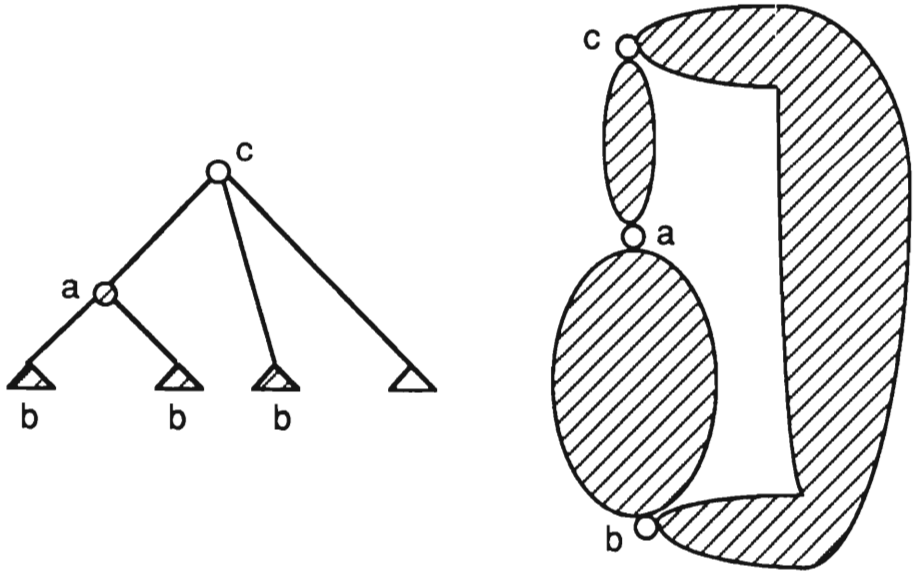


Figure 3.16 - Proof of Lemma 3.17

The last lemma necessary for the identification of the separation pairs of a graph, is the special case of separation pair {1, p}.

Lemma 3.18: The pair {1, p} is a separation pair if and only if the reduction for vertex p ends with Root(T, S) = Root(T) having at least three children.
Proof: If {1, p} is a separation pair, then there must be at least two non-trivial separation classes with respect to {1, p}. Thus, each of these separation classes will contribute a single (virtual) edge for the final P-node. From the definition of st-numbering, there is also an edge (1, p).
The converse follows directly from the statement and using an analogous proof to that given in Lemma 3.15. ❑

The preceding lemmas all concentrate on full nodes in the PQ-tree. From Section 3.1, and from Karabeg [Kar88], we have the following lemma that justifies why we can restrict ourselves to full nodes. The proof of the sufficiency of the lemmas will be proved implicitly by the proof of the algorithm's correctness.

Lemma 3.19: If, for a planar graph G and corresponding PQ-tree T, no full nodes appear throughout the running of the PQ-tree reduction process of T, then G has only one embedding in the plane.

Using Theorem 1.14 and Lemma 3.19, we can conclude that the absence of full nodes during the reduction pass implies that the graph has no separation pairs. Thus, a linear time triconnectivity testing algorithm follows directly. For a planar graph G, we merely proceed with the reduction pass, checking if we match either Template P_1 or Template Q_1 . If no match is found, then we know that G is 3-connected. The extension of the algorithm to produce the split components of G is not as easy. We have to consider the proper maintenance of the labels of P-nodes and the joints of Q-nodes in the PQ-tree. Also, we need to keep a record of the edges which correspond to a particular node of the PQ-tree, so that when we find a split component, we can easily determine the corresponding edges. We call this list the *Edge List* for that node.

The proper maintenance of the labels of P-nodes and the joints of Q-nodes in the PQ-tree is easy. As already discussed, there are two cases when a P-node is inserted into a PQ-tree. After a reduction for a vertex with st-number k, we substitute the full nodes with a single P-node having children being leaves representing edges directed out of k. Here the P-node is assigned label k. Also, during Templates P_2, P_3, ..., P_6 we may possibly introduce a new P-node. In this case we give the P-node the label of its parent P-node. In Templates P_4 and P_6, the joint of the Q-node explicitly shown in the replacement pattern is assigned to be the label of its P-node parent. For Templates P_3 and P_5, where the Q-node replaces the P-node that matched the template, the joint of the Q-node is not yet assigned a value. The justification of these assignments was discussed earlier in this section.

The proper maintenance of the Edges Lists corresponding to a node is not as easy. We obtain the following rules for properly maintaining the Edge Lists.

1. If a pertinent leaf is Root(T, S), then the single edge represented by that node is stored in the new P-node which we add as the leaf's replacement in the vertex addition stage.
2. If a pertinent leaf is not Root(T, S), then we must add the graph edge which the leaf represents to the parent node. A problem occurs when the parent node is the Pseudo Node, which we discuss below.
3. If we merge two Q-nodes, then we merge the two lists of edges together, and assign this new merged Edge List to the new Q-node. Again, a problem occurs when the new Q-node is the Pseudo Node, which we discuss below.
4. If a new Q-node replaces a P-node, this happens when we apply Templates P_3 and P_5, then we assign the Edge List of the P-node to the Q-node. The P-node's Edge List is set to nil.
5. During normal reduction we must add the graph edges which the leaves in the current reduction represent to the Edge List of Root(T, S). Note that, in the replacement pattern, we may change

Root(T, S) (for example Template P_6). In this case, we must add the graph edges to the Edge List of the new Root(T, S) shown in the replacement. Again, a problem occurs when the Root(T, S) is the Pseudo Node, which we discuss below.

The Edge List of a P-node is either empty or contains exactly one element, a Q-node may have any number of edges in the Edge List. Note that Karabeg [Kar89] fails to consider the Pseudo Node case when updating the Edge Lists. For the case when we have a Pseudo Node, we must assign the Edge List to the parent Q-node; but at this stage we do not know the proper parent. In order to keep the algorithm linear, we are unable to search through the empty siblings of the Pseudo Node to an endmost child, and hence obtain the parent. The solution we propose is to keep two lists for any node. The first list is the Edge List that specifies the graph edges which the node represents. The second list is called the *Parent Edge List*. This list specifies the edges which belong to the parent, but that we are unable to add to the parent as yet. Then, when we perform the bubble pass of the algorithm, we check if a node has a non-empty Parent Edge List. If so, and the parent is determined by the bubble pass, then we add this list to the parent's Edge List.

If the parent cannot be determined as yet, then we must delay adding the Edge List to the parent Edge List. Since the parent is a Q-node, it may merge with another parent node (e.g. Template Q_2). It is not critical for the incomplete Edge List to be added, since we note that the proper Edge List is only required for a node when the node is full. Thus, the bubble pass would have assigned the non-empty Parent Edge List to the parent Q-node before the parent Q-node could be matched to a full template.

The complexity of the algorithm remains unchanged if we insert the above modifications to maintain the Edge Lists. The merging of lists is easily performed if we use a simple circular linked list structure. Then, to merge two lists we merely change a few pointers (see Chapter 1 for details).

The pre-processing bubble pass remains exactly as described in Section 2.5, except for the simple check which we discussed above to carry out a merge operation with the parent Edge List. The important changes are in the Template matching pass and in the vertex addition pass stage of the algorithm. We have the following modified template matching algorithm. Note that we omit some detail from the algorithm, in particular we omit the usual steps to update the Pertinent_Leaf_Count. These details may be found in the original algorithm, Algorithm 2.13, in Section 2.5.

Algorithm 3.13: Modified_Reduction_Algorithm(T, S)

```
Queue = empty
for every Leaf ∈ S do
    Add Leaf to Queue
Component_No = 0                    { No split components yet }
Perform_Substitution = false  { No split component at Root(T, S) yet }
while |Queue| > 0 do
    Remove Current from head of Queue
    if Current ≠ Root(T, S)
        then
            if all the kids of Current's parent have all been matched
                then add Current's parent to Queue
    { Attempt to match template to Current }
            if not Template_L1(Current)
                then if  Template_P1(Current)
                    then                { we have a separation pair }
                        Split_Component (Current)
                else
                then if not Template_P3(Current)
                then if not Template_P5(Current)
                then if  Template_Q1(Current)
                    then                { we have a separation pair }
                        Split_Component (Current)
                else
                then if not Template_Q2(Current)
                then        { no templates match - tree is irreducible }
                    T = ∅           { return the null tree }
                    Halt            { graph is non-planar }
        else
    { we are at Root(T,S), attempt to match Root to a root template }
            if not Template_L1(Current)
                then if  Template_P1(Current)
                    then                { we have a separation pair }
                    Split_Component (Current)
                    Perform_Substitution = true
                else
                then if  Template_P2(Current)
                    then                { we have a separation pair }
```

```
            Split_Component (Current)
            Perform_Substitution = true
    else
    then if not Template_P4(Current)
    then if not Template_P6(Current)
    then if not Template_Q1(Current)
        then                { we have a separation pair }
            Split_Component (Current)
            Perform_Substitution = true
    else
    then if not Template_Q2(Current)
    then if not Template_Q3(Current)
        then    { no templates match - tree is irreducible }
            T = ∅           { return the null tree }
            Halt            { graph is non-planar }
end
```

The procedure Split_Component performs the splitting operation, and performs the appropriate additions of a virtual edge as well. The variable Perform_Substitution is a boolean flag which is used to indicate that we have found a split component at Root(T, S). This is necessary because, in the vertex addition stage of the algorithm (presented below), we must ensure that we check if Lemma 3.16 is satisfied (i.e. if the pair is {a, b}, then we may have only one vertex adjacent to b with greater st-number, and in this case this vertex and a also form a separation pair).

Note that Karabeg's algorithm [Kar89] fails to provide a split when Template P_2 is successfully matched. When Template P_2 is matched, we must split the graph because we now have a full node (i.e. the new Root(T, S)), and the node satisfies Lemma 3.15.

With every PQ-tree node, we associate another label, which takes on the values 'virtual' or 'non-virtual'. If, during a reduction for vertex b, we encounter a full P-node (Q-node) with label a (joint a), after outputting the split component we have found, we replace the maximal subtree rooted at the full P-node (Q-node) with a leaf node representing the virtual edge e = a b, and having label 'virtual'. All nodes not created by the above process receive the label 'non-virtual'.

When calling Split_Component(Current), and Current is a P-node, then we note that, the variables Non_virtual_found and Virtual_Found

indicate whether the child of the P-node found so far is virtual or not. The variable Virtual_Label indicates the label of the virtual edge represented by the child in question. We now describe the Split_Component algorithm.

Algorithm 3.14: Split_Component (Current)

Let the Label {joint} of Current be a, and suppose that we are reducing for vertex b.

```
if Node_Type (Current) = P-node
    then
        { Initialise Variables }
    Non_Virtual_Found = false   { no non-virtual child yet }
    Virtual_Label = 0           { no virtual label yet }
    Virtual_Found = false       { no virtual child yet }
        { we now gather data for the first child }
    If first Child is not virtual
        then
            Non_Virtual_Found = true
        else
            Virtual_Found = true
            Virtual_Label = Child's Virtual Label

    For every other Child of Current do
        if  Virtual_Found = true
            then            { we have found a triple }
                Component_No = Component_No + 1
                Output a triple bond {a, b} with
                    (Virtual_Label,
                    Component_No {a, b},
                    This Child status)
        else            { Non_Virtual_Found must be true }
            then            { we have found a triple }
                Component_No = Component_No + 1
                Output a triple bond {a, b} with
                    (Graph Edge,
                    Component_No {a, b}
                    This Child status)
                Non_Virtual_Found = false
        Virtual_Found = true
        Virtual_Label = Component_No
    end {of the for statement }
    If Edge List of Current ≠ nil       { Lemma 3.17 }
        then    { we have a case of Type 2a - vertex of degree 2 }
            Let the single edge in the Edge List be {c, a}
            Output as a triangle
                (the edge in the Edge List,
                Component_No virtual edge {a, b},
                Component_No + 1 virtual edge {c, b} )
            Component_No = Component_No + 1
```

```
        else        { Current is a Q-node }
                { Subtree rooted at Current represents a split component }
            Output split component rooted at Current
                specified by Edge List of Current
                    and edges represented by full leaves
            Output virtual edge Component_No
                    Component_No = Component_No + 1

    If Current has no Siblings      { must have a P-node parent }
        then                        { Check for Lemma 3.17 case }
            if Label of Parent ≠ Label of Current
                then
                    Let Label of Parent be c.
                    Output as a triangle
                        (edge (c, a)        { which may be virtual }
                        Component_No virtual edge (a, b),
                        Component_No + 1 virtual edge (c, b) )
                    Component_No = Component_No + 1
        Change Current to a virtual leaf with virtual label Component_No
```

end

Firstly, notice that we do not correctly check for Lemma 3.18. If Current ≠ Root(T, S) and the label or joint of Current = 1, and we are reducing with respect to vertex p (so Current must be a child of Root(T, S)), then, by Lemma 3.18, Current may <u>not</u> create a split component. However, at the very next stage we would either output the remaining edges in Edge List of Current the tree as the remaining split component (so Root(T) only has two children), or, if Root(T) had more than two children, we would output the split component triple bonds. Thus, in the latter case this produces the split components correctly, and in the former case we merely replace the virtual edge 1 → p in the second to last split component to be a tree edge (since we know that edge 1 → p is always in the graph), and we delete the last split component found (i.e. the triple bond 1 → p).

Further, note that we know that a full P-node only has leaf edges (either virtual or graph edges), because the algorithm would have already discovered the full children P-nodes or Q-nodes, and would have replaced them with virtual edges. Now, every two full children (virtual or graph edges) induce a split component. Note that Karabeg incorrectly considers each child as a triple bond, which is certainly not the case.

In an analogous manner to Hopcroft and Tarjan's algorithm in Section 3.2, we need to check for chains of P-nodes which only have a single child which are created, since they give rise to Type 2a separation pairs. There is a special check for a P-node which does not have an empty Edge List. If this is the case, then the Edge List could only have been created when the P-node was originally a leaf. Since a P-node never has more than one edge in its Edge List, we know that there is exactly one edge in the Edge List, and further that a chain is present (see Figure 3.16 and Lemma 3.17). We note that the chain is implicitly represented. Karabeg provides no such method of finding chains arising from single leaves added in the Vertex Addition stage, which we find by checking the Edge List of the P-node. If it is not empty, then we have discovered a split component which is a triangle.

Note that, contrary to Hopcroft and Tarjan, we do not have to check for multiple edges, since these edges would be found at a later stage in the algorithm at a full P-node. The checking for Lemma 3.17 is performed in the algorithm as a last check after reduction is complete.

The last algorithm to consider is the modified Vertex Addition stage of the algorithm.

Algorithm 3.15: Modified_Vertex_Addition (Current)

We have reduced T with respect to vertex b, now we perform the vertex addition

```
if not Perform_Substitution
    then        { Root (T, S) must be a Q-node }
        Replace sequence of full children and descendants by a
        P-node, labelled b,  with children the edges directed out of vertex b

    { Note from Algorithm 3.14 and 3.13 that, if Perform_Substitution
      is true, then separation pair {a, b} is found }
    else
        Replace Root(T,S) and its descendants by a P-node with children
            the edges directed out of vertex b
        If there is only one child edge, to a vertex d, to add
            then        { Lemma 3.16 is satisfied }
                        { See Figure 3.15 }
                Output as a triangle
                    (edge b d
```

```
                    Component_No virtual edge {a, b},
                    Component_No + 1 virtual edge {a, d} )
                Component_No = Component_No + 1
                Current = virtual edge leaf {a, d} with label
        Component_No
```

end

All that remains now is to prove that the triconnectivity algorithm does indeed find the split components of a plane graph G. But, first we need a lemma.

Lemma 3.20: Let {a, b} be a separation pair ($a \neq 1$ and $b \neq p$) such that at least one of the separation classes of G, with vertices v such that $a \leq v \leq b$, with respect to {a, b} is 3-connected. Then, during the reduction for vertex b, a full P-node with label a, or a full Q-node with joint a, must be found.
Proof: The proof follows directly from the fact that, since {a, b} is a separation pair, we have, from Lemma 3.14, that we must obtain one of the separation classes of G with respect to {a, b} during the reduction for vertex b. Lemma 3.19 then concludes the proof, since {a, b} is a separation pair. ❑

Theorem 3.5: The modified bubble pass algorithm, the modified reduction algorithm (Algorithms 3.13 and 3.14), and the modified Vertex Addition algorithm (Algorithm 3.15) correctly find the split components of a planar graph G, and do so in linear time.
Proof: The algorithm will only split a graph if, during the reduction for a vertex b, a full node with joint (or label) a is found. By Lemma 3.15 and Lemma 3.18, then {a, b} is a separation pair. It suffices then to show that each of the separation classes found in Algorithm 3.14 and 3.15 do not contain any separation pairs.
Suppose, to the contrary, that a separation class S found by the algorithm does indeed contain a separation pair {c, d}. It is easy to verify that all triangles are discovered correctly (either by Lemma 3.17 or Edge_List $\neq \emptyset$), and so we may assume that the separation classes with respect to {c, d} with vertices $c \leq v \leq d$ are not triangles. If one of the separation classes with respect to {c, d} with vertices $c \leq v \leq d$ is 3-connected, then by Lemma 3.20, a split component would have been discovered by the algorithm. If

none of the relevant separation classes are 3-connected, then each separation class with vertices $c \leq v \leq d$ contains another separation pair. By applying the argument recursively we eventually obtain such a 3-connected separation class, and hence a contradiction.

The complexity is easy to determine. We have, from Section 3.2, Lemma 3.5, that the total number of edges in the split components is at most 3q - 6. Thus, we add virtual edges to the PQ-tree a limited number of times. Also, we note that to output the split components corresponding to a P-node is of the order of the number of children of the P-node. Similarly, to output the split component corresponding to a full Q-node, is of the order of the number of edges in the split component. Note that the argument employed in Section 2.5 to obtain a linear time complexity is still valid, even though we may now have chains of P-nodes with single children in our PQ-tree T. Since the chain nodes in T are deleted once they are matched, during an S-reduction, $|Unary(T, S)|$ does not grow too large. Thus, the overall complexity of the algorithm is O(p). ❑

We close this section with an example. Suppose that we are reducing the following graph.

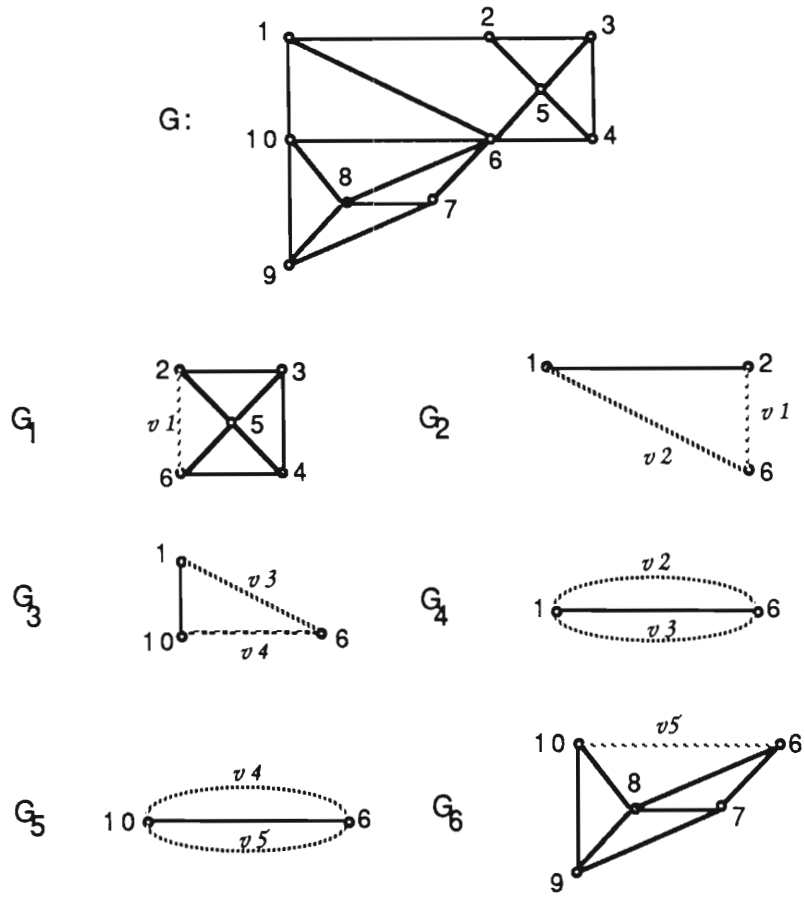


Figure 3.17 - A graph G and the split components G_1, G_2, ..., G_6 of G.

The reduction pass for vertex 2 proceeds as normal, and, after the vertex addition stage, we obtain the following PQ-tree. The outlined number next to a node indicates the label or joint of that node.

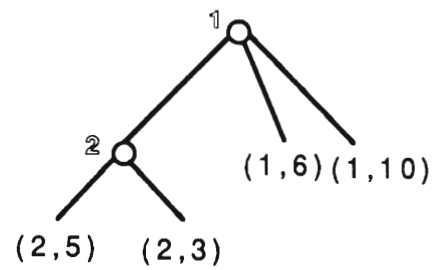


Figure 3.18 - PQ-tree T after reduction for vertex 2

Note that at this stage the P-node with label 2 has an Edge List containing the single edge (1, 2). We now reduce for vertex 3, and after the vertex addition stage, we obtain the PQ-tree shown in Figure 3.19, below.

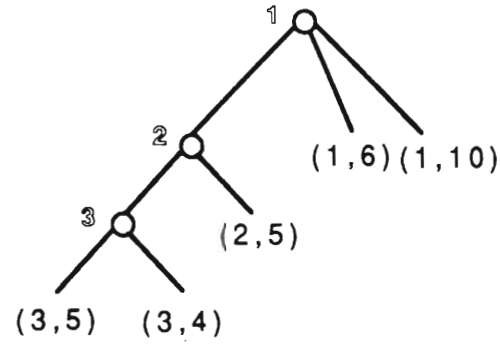


Figure 3.19 - PQ-tree T after reduction for vertex 3

Again, the P-node with label 3 has an Edge List containing the single edge (2, 3). The reduction for vertex 4 is trivial, and we obtain the following PQ-tree, shown in Figure 3.20.

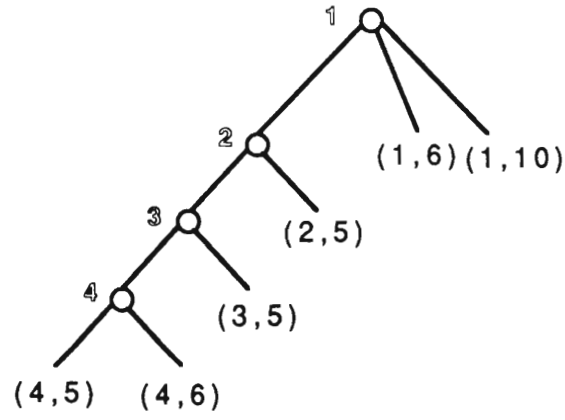


Figure 3.20 - PQ-tree T after reduction for vertex 4

At this stage, the node labelled 4 has an Edge List containing the single edge (3, 4). The reduction for vertex 5 is non-trivial. Templates P_3 and P_5 are triggered once each, and then Template P_4 is triggered for Root(T, S). As discussed after Lemma 3.17, we keep the P-node with label 2. The Q-node child is assigned the joint of 2. The resultant PQ-tree is shown below, in Figure 3.21.

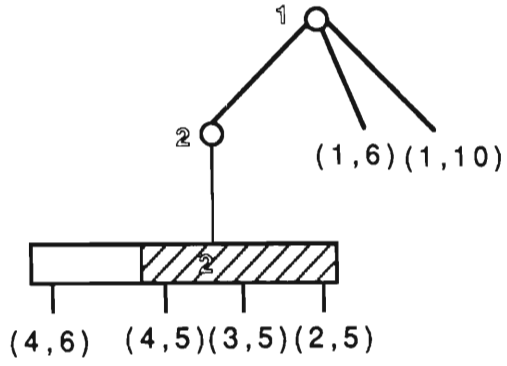


Figure 3.21 - PQ-tree T after reduction for vertex 5

As usual, we group the full children together, add the edges directed out of vertex 5. In this case we have a single child, the edge (5, 6). The Edge List of the full children is added to the Q-node parent. The Edge List of the Q-node is now {(2, 3), (3, 4), (3, 5), (4, 5), (2, 5)}. Now, the reduction for vertex 6 proceeds, and we find that there is a full Q-node during the reduction pass (Figure 3.22(a)). So, we output the Edge List, the full children edges and the virtual edge (2, 6) as a new component. The Q-node becomes a virtual leaf, with label 2 and leaf value (2, 6). Now, we match the parent P-node to Template P_1, but, since the P-node parent has a single child, no triple bonds are found. Figure 3.22(b), below, shows the PQ-tree. The check for a non-empty edge list at the node labelled 2 succeeds (Edge List = {(1, 2)}, and so we may output the triangle {(1, 2), (2, 6), (1,6)}. The virtual edge (1, 6) replaces the P-node and the virtual child.

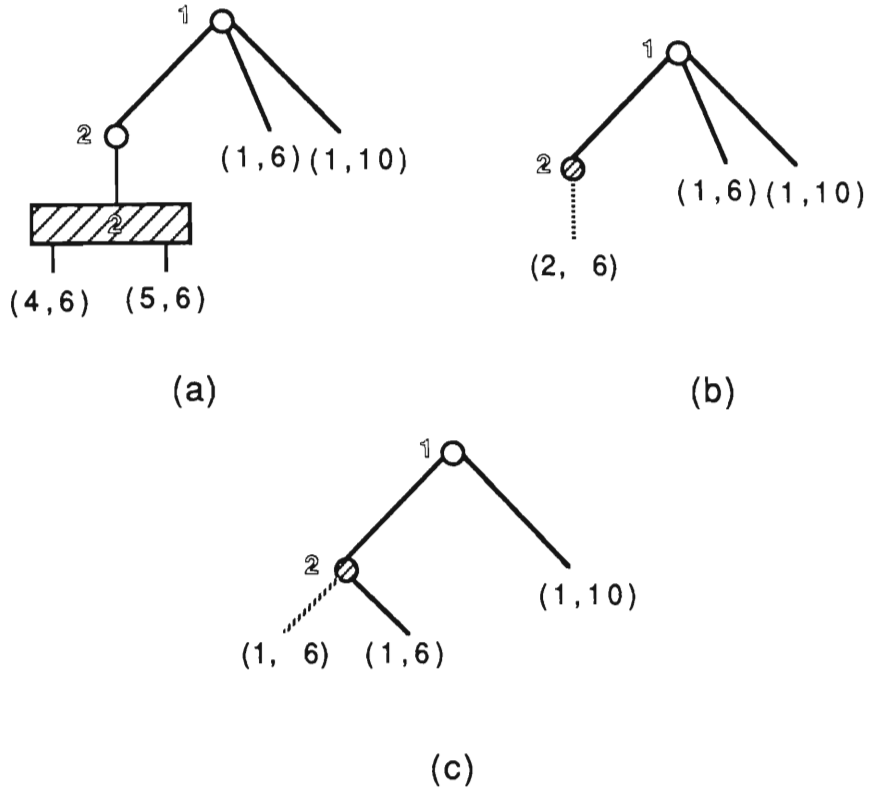


Figure 3.22 - PQ-tree T after reduction for vertex 6

We then continue with the reduction pass for vertex 6. The Root(T, S) matches Template P_2, and we have another split component. The situation is described in Figure 3.22(c). We output a triple bond with virtual edge (1, 6), graph edge (1, 6) and another virtual edge (1, 6). We replace the full P-node with a virtual leaf with label 6. We then proceed to the vertex addition stage. The virtual leaf becomes a virtual P-node, with Edge List {(1, 6)}. Then, we add the edges incident from vertex 6, namely (6, 10), (6, 7) and (6, 8), to the tree. The reduction for vertex 7 is easy (just a single edge), and we replace the leaf (6, 7) with a P-node and the edges incident from 7. The Edge List of the P-node is {(6, 7)}. The resulting graph is shown, below, in Figure 3.23.

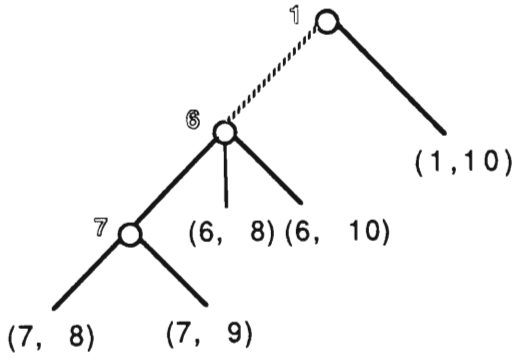


Figure 3.23 - PQ-tree T after reduction for vertex 7

The reduction for vertex 8 is non-trivial. Templates P_3 and P_4 are triggered. The resulting Q-node is assigned the label of the P-node parent, namely 6. The PQ-tree after the vertex addition stage for vertex 8, is shown in Figure 3.24, below. Note that the Edge List of the Q-node is {(6, 7), (7, 8), (6, 8)}.

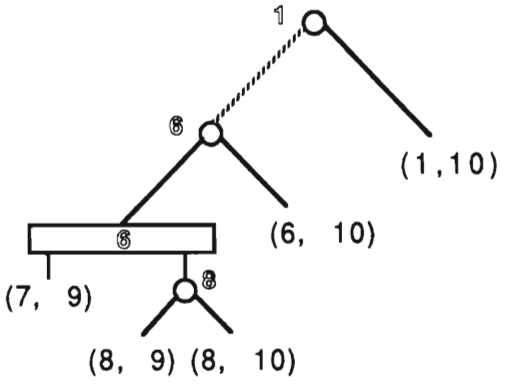


Figure 3.24 - PQ-tree T after reduction for vertex 8

Reduction for vertex 9 is non-trivial, but fairly simple. Template P_3 is triggered for the P-node parent of leaf (8, 9). Template Q_2 is triggered for Root(T, S). We replace the full children by a new leaf node, edge (9, 10). The Edge List of the Q-node is now {(6, 7), (7, 8), (6, 8), (8, 9), (7, 9)}.

The reduction for vertex 10 is non-trivial, and yields the last split components. Firstly, we encounter a full Q-node, so we output the split component corresponding to the Q-node. The Edge List of the Q-node is

{(6, 7), (7, 8), (6, 8), (8, 9), (7, 9)} and we have full children (8, 10), (9, 10). Thus, we output the component with edges (6, 7), (7, 8), (6, 8), (8, 9), (7, 9), (8, 10) and (9, 10), and with virtual edge (6, 10). We replace the Q-node with a virtual leaf (6, 10). The reduction then proceeds to the P-node parent, and we get another full node, this time a triple bond. Figure 3.25, below, shows the PQ-tree so far.

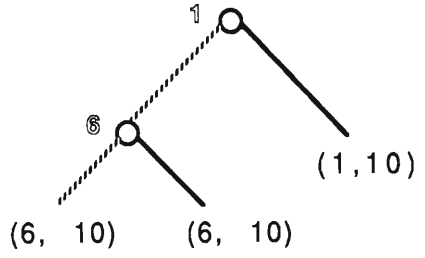


Figure 3.25 - PQ-tree T during reduction for vertex 10

We output the triple bond with edges (6, 10), and Lemma 3.17 is now satisfied. We have a P-node parent with different label to the current P-node. Thus, we output a triangle with edges the virtual edge (1, 6), virtual edge (6, 10), and a new virtual edge (1, 10). Lastly, we observe that, since the full P-node of Root(T, S) has only two children, the node does not induce another split component. But, as in the discussion following Algorithm 3.15, the algorithm initially outputs another triple bond (1, 10). We modify the last virtual edge (1, 10), and note that it is a graph edge, and we discard the last triple bond split component.

As a closing remark, we note that Karabeg extended the original template set to include four new templates which cater for non-planar graphs. Since the algorithm described above only deals with full nodes, it is possible to modify the algorithm to extend it to triconnectivity testing of (non-planar) graphs. We defer discussion of these extra templates until Chapter 4.

Section 3.4

Drawing a Graph in the plane

As discussed at the beginning of this chapter, we shall look at two linear algorithms by Chiba, Yamanouchi and Nishizeki [CYN84]. Although there are simpler drawing algorithms (for example Tutte [Tut63]), the algorithm complexity is typically $O(p^3)$. The problem of producing satisfactory circuit layouts, VLSI (Very Large Scale Integration) often implies graphs of the order 100 000 or more. Thus, we restrict the drawing algorithms under consideration to linear time algorithms. We follow the approach by Nishizeki and Chiba [CN88] for details. For convenience, we shall repeat some of the definitions and terms pertaining to the drawing algorithms.

We say that a planar graph G has a *convex drawing*, if there is a drawing of G on the plane so that all edges of G are represented by straight lines, so that no two lines meet, except at their endpoints, vertices are represented by terminal points, and the boundary of each region in the drawing is a convex polygon. Note it is possible for a vertex not to correspond to an apex of the convex polygon. This happens if the two edges incident with a vertex x lie in a straight line. In this section the two algorithms we present will, firstly determine if a convex drawing of a graph G is possible, and secondly produce a convex drawing of G if it is possible.

More exactly, we say that a convex polygon S^* of a facial cycle S of a plane graph G is *extendible* if there exists a convex drawing of G having S^* as the outer facial cycle of G. A facial cycle S is extendible if and only if S has an extendible convex polygon S^*. Given a planar graph G and an extendible convex polygon S^* of a facial cycle S, the first algorithm that we discuss will provide a convex drawing of a planar graph G in linear time. The second algorithm tests if G has a convex drawing, and, if so, outputs the extendible convex polygon S^* of a facial cycle S, also in linear time.

Producing a Convex drawing from an extendible cycle

The first algorithm is based on a result by Thomassen [Tho80]. His result is in turn based on a result by Tutte [Tut60], which provides a necessary and sufficient condition for a convex polygon to be extendible. We have the following lemma from [Tho80]. Note that a more rigorous proof of the lemma would require topological arguments which are beyond the scope of this thesis, so we merely sketch the proof.

Lemma 3.21: Let G be a 2-connected plane graph with outer facial cycle S, and let S^* be a convex polygon of S. Let the paths of S which correspond to sides of S^* be $P_1, P_2, \ldots P_k$. Then, S^* is extendible if and only if Condition 1, below, holds

Condition 1

(a) For each vertex $v \in V(G) - V(S)$, having degree at least three in G, there exists three paths, disjoint except for v, each joining v and a vertex in S.

(b) The graph G - V(S) has no component C, such that all the vertices on S, adjacent to vertices on C, lie on a single path P_i; and no two vertices in each P_i are joined by an edge not in S.

(c) Any cycle of G with no edge in common with S has at least three vertices of degree at least 3.

Proof: Suppose S^* is extendible. To show that Condition 1 follows we consider Figure 3.26. In Figure 3.26(a) we illustrate a situation for which Condition 1 (a) does not hold. The vertex v is any vertex which is common to the two shaded polygons in the drawing. We cannot draw the smaller polygon convex, and so S^* is not extendible. A situation for which Condition 1 (b) does not hold is illustrated in Figure 3.26(b). Both points are illustrated. We let the connected component C be the shaded region in the drawing. If two vertices on a path P_i are joined by an edge not in S (i.e. either small arch in the drawing), then it is easy to see that it is not possible to extend S^*. The other point is seen if we then suppose that the two arches have other vertices on them, then obviously $C \cup S$ does not

have a convex drawing. Lastly, we may easily observe the necessity of Condition 1 (c) by considering, in Figure 3.26(c), a situation for which Condition 1 (c) does not hold. Clearly both shaded polygons may not be drawn convex if the cycle is to be drawn convex as well. The sufficiency of the lemma will follow implicitly from the algorithm. ❑

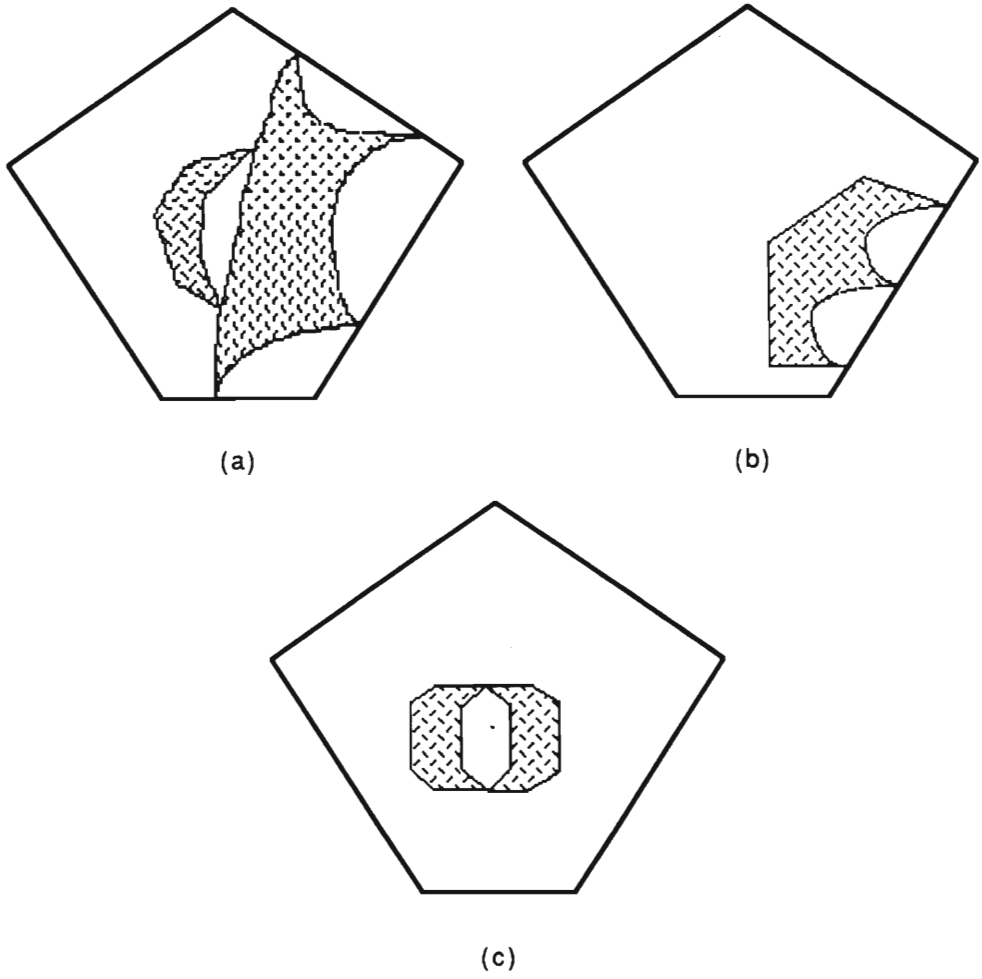


Figure 3.26 - Proof of Condition 1

Although Condition 1 does allow for vertices of degree 2, we assume, for the rest of the section, that each vertex not on S has degree at least 3. We say that a convex polygon S^* of a cycle S is *strict*, if every vertex $v \in V(S)$ is an apex of S^*. We draw the outer facial cycle S as a strict convex polygon. Then, consider every path P which is maximal with respect to the property that every internal vertex is not on S and has degree 2. The path P must necessarily be drawn as a straight line. Thus, we replace P with a

single edge connecting the two endpoints, to form a new graph G'. Once G' is drawn convex, we then proceed to insert the vertices of degree 2 at equally spaced distances between the two endpoints of the path.

We note that if the minimum degree of G is greater than 2, then Condition 1(c) appears to be redundant, however, we need Condition 1(c) in the recursive step, where subgraphs may contain vertices of degree 2. We assume for the rest of the sub-section that we have an embedding of a planar, 2-connected graph G, and an extendible cycle S, with every vertex not on S having minimum degree greater than 2. An embedding of G may be obtained easily from the results of Section 3.1.

The algorithm proceeds by drawing the outer facial cycle S, and then recursively drawing the components of G - V(S). We delete an arbitrary apex v of S^*, together with the edges incident with v. For the resulting graph we determine the blocks $B_1, B_2, ..., B_k$ ($k \geq 1$). Then, we determine, for each block B_i, ($1 \leq i \leq k$), for the outer facial cycle S_i of B_i, a convex polygon S_i^*, so that Condition 1 is still satisfied. Then, we recursively apply the algorithm to each block B_i with convex polygon S_i^*. Figure 3.27 illustrates the idea of the algorithm. The shaded, thick lines are the edges incident with cut-vertices of G - v. The outer facial cycles of the blocks B_i are drawn as thick, solid lines.

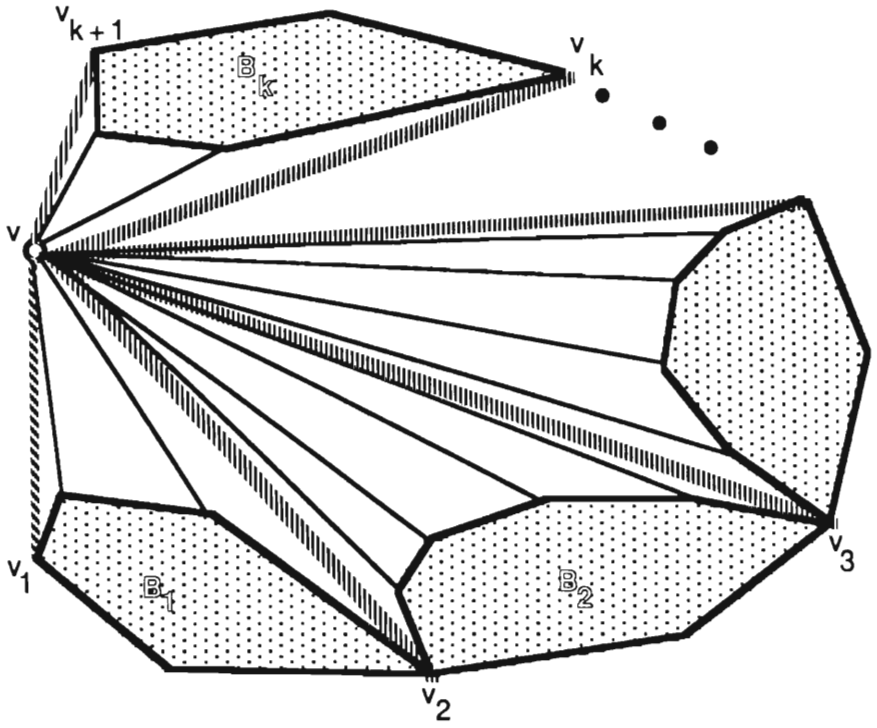


Figure 3.27 - Illustrating the recursive drawing algorithm

Notice that, for each block B_i, with extendible cycle S_i^*, the apexes of S_i^* are the vertices adjacent to v, together with the old vertices of S in G - {v}. Note that the interior of each triangle which has v as an apex is necessarily bounded by a convex polygon, since we only make vertices adjacent to v apexes. Now, if we can draw each block B_i as a convex polygon, it follows since each triangle which has v as an apex is drawn convex that we obtain a convex drawing of G. Thus, by successfully carrying out the recursive step of drawing a convex drawing of each block B_i, we will ensure that the graph G is drawn in the plane as a convex drawing. Because G is 2-connected, we note that the removal of v from G does not disconnect G. Let the two vertices adjacent to v, on S, be v_1 and v_{k+1}. Since there are no vertices of degree 2 in G, and also by Condition 1(a), we note that every cut vertex v_i of G - {v} is necessarily on S. Let the cut-vertices of G' be v_i ($2 \leq i \leq k$), so that $v_1 \in V(B_1)$, $v_{k+1} \in V(B_k)$, and $v_i \in V(B_{i-1}) \cap V(B_i)$, for $2 \leq i \leq k$.

From Figure 3.27, we see that the placement of the extendible cycle S_i^* is also important. Each S_i^* must not overlap with the neighbouring cycles S_j^*. Thus, we restrict the placement of S_i^* to lie within the triangle with apexes v, v_i and v_{i+1}, and in such a manner that S_i^* is a convex polygon. We may now present the drawing algorithm. We assume that the extendible polygon S^* of S is drawn.

Algorithm 3.16: Convex_Drawing (G, S, S^*)

{ Given an extendible convex polygon S^* of S, draw G }

```
If | V(G) | ≤ 3
    then G is drawn
            { must be, since S is a cycle and all vertices of S are placed }
    else
        Select arbitrary apex v of S*
        G' = G - v
        Find the blocks Bi, (1 ≤ i ≤ k), of the graph G'

        { Now set up variables as in Figure 3.27, and note following }

        Let the two vertices adjacent to v, on S, be v1 and vk+1
        Let the cut-vertices of G' be vi (2 ≤ i ≤ k), so that
        v1 ∈ V(B1), vk+1 ∈ V(Bk), and vi ∈ V(Bi-1) ∩ V(Bi), (2 ≤ i ≤ k)

        { Now we perform the drawing }
        For each block Bi do
        Draw convex polygon Si* of the cycle Si of Bi as follows :

            Determine the positions of the vertices vm ∈ V(Si) - V(S)
                if vm is adjacent to v
                    then
                        Place vm as an apex of Si*
                    else
                        Place vm so that it lies on a
                            straight line segment of Si* .
            Note : vm must lie entirely in the triangle v, vi, vi+1
            The resulting polygon Si* must be a convex polygon.

        { Recurse to complete drawing of Bi }
        Convex_Drawing (Bi, Si, Si* )
end
```

Theorem 3.6: Let G be a plane, 2-connected graph, with outer facial cycle S, and let S^* be an extendible convex polygon of S. Algorithm 3.16 extends S^* into a convex drawing of G.

Proof : As usual, let v be an arbitrary apex of S^*, and let the blocks of G - {v} be $B_1, B_2, ..., B_k$ ($k \geq 1$). For the outer facial cycle S_i of each B_i, ($1 \leq i \leq k$), let a convex polygon of S_i be S_i^*. As mentioned before, all the triangles with apex v in Figure 3.27 are convex, since we choose the apexes of each S_i^* which belong to S_i^* - S to be vertices adjacent to v. Now, we note that if a block B_i violated Condition 1(a), then so would G, since then G would contain a vertex w of degree at least 3 with at most two paths, disjoint except for w, joining w and vertices of S. Condition 1(b) is harder to see. Suppose B_i - $V(S_i)$ has a connected component C, such that all the vertices on S, adjacent to vertices on C, lie on a single path P_i of S_i^*. Then, suppose P_i is a subpath of a path P_j of S, then G would have violated Condition 1(b). If P_i is not a subpath of a path P_j of S, then, by assumption, there are no edges from C to a path P_j of S. However, then consider the graph C + P_i. In G, Condition 1(a) would have been violated (see Figure 3.28, below). Since we are only dealing with graphs with minimum degree 3, Condition 1(c) is redundant, and so the theorem is proved. ❑

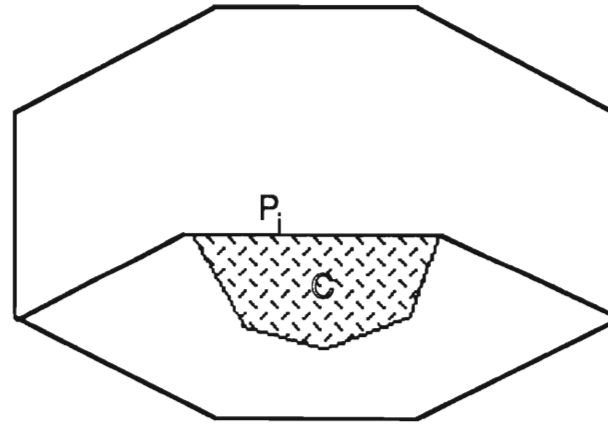


Figure 3.28 - Illustrating Theorem 3.6

There are some details in Algorithm 3.16 which need to be elaborated upon before we can justify a linear time and space complexity of the algorithm. In particular, a problem occurs when finding the blocks of G - v. Although the Depth-First Search (DFS) does provide an efficient

method for determining the blocks of a graph, we need a technique which allows for repeated applications. Since G satisfies Condition 1, all cut-vertices of G - v are on S. To determine the cut-vertices, we merely have to traverse the adjacency list of v. If a vertex w, adjacent to v, is on S, then w is a cut-vertex. Observe also that these are the only cut-vertices. Suppose that w is in two blocks, B_i and B_j, and that A is the embedding of G. We assume that each adjacency list of A is a doubly linked list. Further, if a vertex w is adjacent to v, then in A(v), the reference to w is accompanied by an extra reference, called *Other_Edge*, to the element in A(w) which refers to v. Thus, tracing out a region in an analogous manner to the Rotational Embedding Scheme is easy, if we use *Other_Edge*. If we start at a vertex v, we merely obtain the reference to v in A(w), and then select the next element in A(w) as the next vertex along the boundary of the region and repeat.

As a first approximation to attempt to find the blocks of the graph, for every cut vertex w, we add a new vertex w_v to the graph. We split A(w) into two groups, such that one group consists of all edges incident to vertices in B_i, and the other group consists of all edges incident to vertices in B_j. The first group is now A(w), whilst the second group is $A(w_v)$. Of course, to be able to do this we need to know what the vertices of B_i and B_j are. For every vertex z in B_j adjacent to w, we then update A(z) to reflect that z is now adjacent to w_v. The problem with this approach that it is not clear how we can obtain an implementation complexity of O(p). We can scan through A(w) in this manner O(p) times (each time we select a vertex v for deletion, this vertex could be adjacent with w, and A(w) may contain exactly one edge after such a split).

The solution we propose is conceptually and practically clearer. We make no effort to physically find the blocks of the graphs. Instead, we keep, for each vertex $v \in V(G)$, two extra fields *Cycle_Clockwise* and *Cycle_AntiClockwise*, which specify the edge elements in A(v) which are incident to v and from v respectively, as we proceed in a clockwise direction about v around the current extendible cycle S. Thus, all edges in A(v) between Cycle_Clockwise and Cycle_AntiClockwise, proceeding in a

clockwise direction, are not considered (effectively, they have been deleted).

Now, when we determine the original cycle S, the fields Cycle_Clockwise and Cycle_AntiClockwise of each $v \in V(S)$ are set accordingly. When recursively drawing a graph G, with a selected vertex v, we start at the edge specified by Cycle_AntiClockwise, which is incident to a vertex w, say. Note that $w \in V(S)$. Each time we encounter a vertex which is adjacent to v, we store it in a list called *Apexes*. Each time we encounter a vertex distinct from v (whether or not it is adjacent to v), we store it in a list called *Cycle_Elements*. Simultaneously, we update the fields Cycle_Clockwise and Cycle_AntiClockwise as we proceed along the cycle region. We repeat this procedure, moving along the edges incident with v, until we encounter another cut-vertex z. We save the old value of Cycle_AntiClockwise for z. $V(S_i)$ is set to Cycle_Elements. Let T_i^* consist of all the elements of Apexes - {w, x}.

Then, we call a drawing routine to place the elements of T_i^* in the triangle v, w, z in such a manner that S_i^* is convex. Next, we recursively call the drawing algorithm to draw a new graph, with Cycle_Elements representing the new cycle, and Apexes representing the apexes of the convex polygon. Notice that we do not have to pass the entire cycle S_i in the recursive stage. The cycle S_i is implicitly represented by the Cycle_Clockwise and Cycle_AntiClockwise fields. Indeed, all that is necessary is to pass a single vertex on S_i. Upon return, we reset the value of Cycle_AntiClockwise for z to the saved value, and continue traversing edges incident to v, and thereby building other cycles S_j. We stop traversing edges incident to v when we reach the edge specified by Cycle_Clockwise. Figure 3.29, below, illustrates the start of the process.

Now, we repeatedly trace out regions in a clockwise manner by using Other_Edge fields until we encounter another vertex of S. The boundaries of these regions are all in the shape of a triangle, and have v as one of the three apexes. The other two apexes are vertices adjacent to v. If there are further vertices on the boundary of this region, they must lie on the straight line which joins the two apexes on the boundary which are

adjacent with v. See Figure 3.29. In Figure 3.29, we denote the direction of the Cycle_Clockwise and Cycle_AntiClockwise fields by a c and an a respectively.

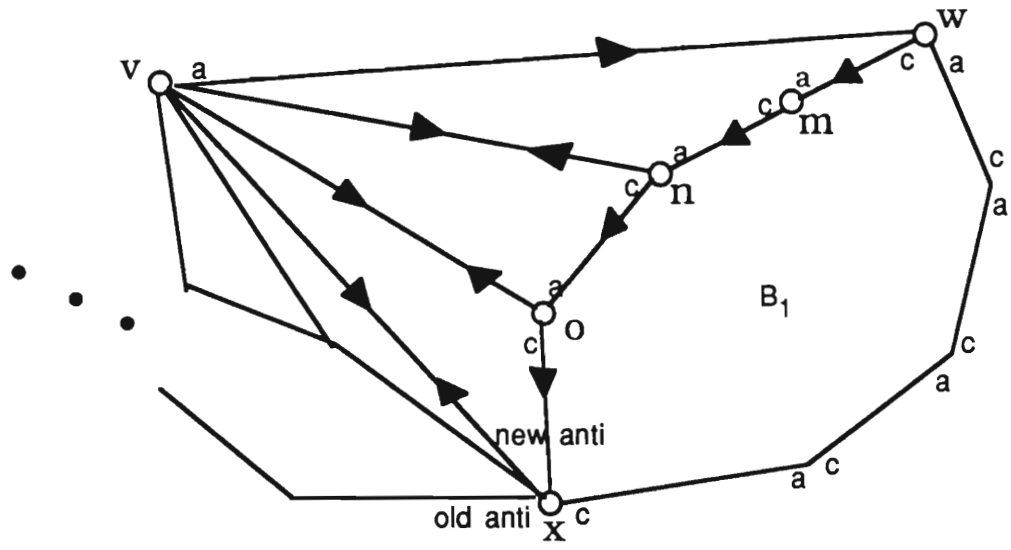


Figure 3.29 - Using Cycle_Clockwise and Cycle_Anticlockwise elements

We denote the saved and new anticlockwise values for x by Old_Anti and New_Anti respectively. Now, Cycle_Elements = {w, m, n, o, z}, and Apexes = {w, n, o, z}. We need one last field before we introduce the detailed algorithm. For each vertex we associate the boolean field *Deleted* which specifies if the vertex has been selected by the Algorithm for deletion before. Initially, all vertices are marked with Deleted = false. Algorithm 3.17, below, gives the detailed algorithm.

Algorithm 3.17: Convex_Drawing (G, S)

{ Given an extendible cycle S, draw the subgraph
of G bounded by the cycle }

{ All vertices of S have been placed in the plane already }

```
Let v = first element of S.
If Deleted(v) = true
    then
        exit    { since the vertex has been selected for deletion earlier }
    else
                { conceptually now, G' = G - v }
```

```
Deleted (v) = true
Let e = Cycle_AntiClockwise (v)        { start edge }
repeat

        { Divide the graph G' into blocks B_i, (1 ≤ i ≤ k) }

    Let Start_Vertex = vertex incident to e different from v

        { initialise - nothing in the block cycle yet }
    Apexes = ∅
    Cycle_Elements = ∅
    w = Start_Vertex

    repeat                  { trace out regions }
                { get edge in A(w) }
        Current_Edge = Other_Edge (e)
                { move e along A(v) }
        e = clockwise edge in A(v) after e
        repeat  { trace out the region }
                    { see Figure 3.29 }

{ Anticlock(e) gets next edge as we proceed around, in anti-
    clockwise direction the vertex that e is incident from }

            Current_Edge = AntiClock (Current_Edge)
            Cycle_Clockwise (w) = Current_Edge

            Next_Vertex = vertex incident to Current_Edge
                { get edge in A(Next_Vertex) }
            Current_Edge = Other_Edge (Current_Edge)
            Cycle_AntiClockwise (Next_Vertex) = Current_Edge
                { check if region complete }
            if Next_Vertex = v
                then    { w must be an apex }
                    Apexes = Apexes ∪ {w}
            Cycle_Elements = Cycle_Elements ∪ {w}
            w = Next_Vertex
        until (w = v)
            { i.e. we have traced out a region }

    until vertex z ≠v incident to e has been placed already
                { i.e. until another cut-vertex }

{ Now, we perform the drawing and place new vertices }
    Draw (Apexes, Cycle_Elements, v, Start_Vertex, z)
{ Recurse to complete drawing of B_i }
    Convex_Drawing (G, Cycle_Elements)
until e = Cycle_Clockwise (v)
```

end

It is easy to see, since an edge is on the boundary of at most two regions, and Algorithm 3.17 scans each edge at most twice, that Algorithm 3.17 has complexity O(q) without the call to the procedure Draw.

All that remains now is to introduce a drawing procedure. There are two stages to the drawing procedure, namely the placement of the apexes and the subsequent placement of the vertices in Cycle_Elements excluding Apexes. The latter stage of the algorithm is relatively easy. We merely space the vertices out at even distances between the two endpoints which have been drawn already. There are always two such endpoints where the positions have already been determined, since, from Figure 3.29, w and z are already placed. The placement of the apexes of S_i^* so that S_i^* is convex is not as trivial.

Figure 3.30(a), below, shows the standard system of axes. Let the apexes of the triangle bounding S_i^* be, as usual, v, w and z. From now on, when we refer to any vertices, we refer to their positions on the screen. We denote v. x and v. y to be the x and y screen coordinates of a vertex. The first operation we perform, is a shifting of the axis. We shift and rotate the axis so that w and z are on the x-axis (i.e. y = 0). Figure 3.30(b), below, shows the resultant system of axis.

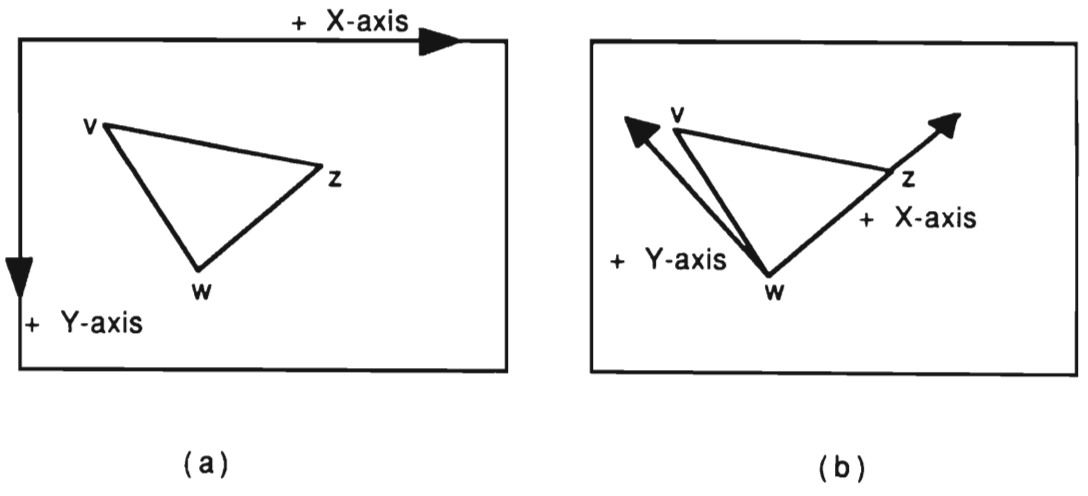


Figure 3.30 - (a) Standard Screen Axis; (b) Transformed Screen Axis

The advantage of the rotation of axis is the corresponding generalisation of the algorithm which accompanies it. The second operation we perform is another simplification. We let *New_Peak* equal v. Then, we shift New_Peak so that New_Peak. x lies between w. x and z. x, as follows. If New_Peak already lies between the two vertices, then leave it alone. Otherwise, suppose that w. x < z. x. If New_Peak is closer to w, shift New_Peak so that its x-coordinate is 20% of the distance from w to z. If New_Peak is closer to z, then shift New_Peak so that its x-coordinate is 20% of the distance from z to w. Figure 3.31, below, shows the three possible cases.

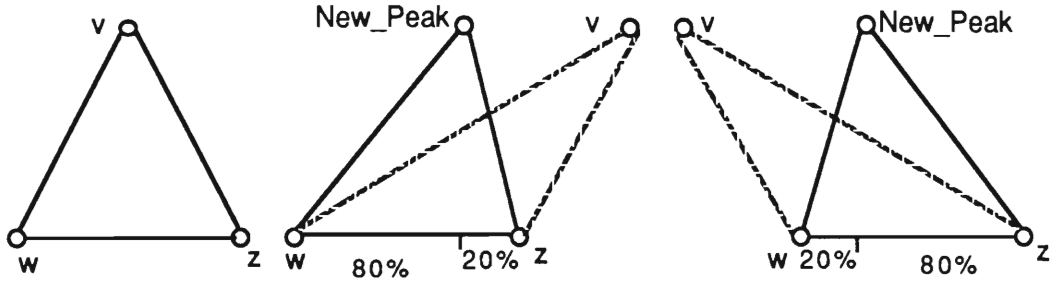


Figure 3.31 - Shifting the peak of the convex polygon

The last transformation before the drawing can begin, is to ensure that New_Peak lies in the region bounded by the triangle v, w, z. This is done by a simple modification of the value of New_Peak. y via a solution of the equation for the line vw or vz (depending on whether z or w is closer to v respectively) when the x coordinate value for the line equals New_Peak. x.

Suppose we have n apexes in Apexes excluding {w, z}. Then, the algorithm considers $\left\lfloor \frac{n}{2} \right\rfloor$ of the vertices in Apexes - {w, z} at a time. The first $\left\lfloor \frac{n}{2} \right\rfloor$ are placed such that they are equally spaced (in terms of x coordinates) between w. x and New_Peak. x, whereas the last $\left\lfloor \frac{n}{2} \right\rfloor$ are placed such that they are equally spaced (in terms of x coordinates)

between New_Peak. x and z. x. If n is odd, then the middle apex is positioned after the first $\lfloor \frac{n}{2} \rfloor$ apexes have been positioned. All that remains is to determine a suitable y increment so that the resulting polygon remains in the bounds of the triangle v, w, z, and is convex.

Let $k = \lfloor \frac{n}{2} \rfloor$. Let the first half of the vertices to be placed be $v_1, v_2, \ldots, v_k$. Further, let $v_0 = w$. To find the value of the initial increment, we merely compute the gradient m and intercept c of the line w, New_Peak, and assign $v_1.y = (m * v_1.x + c)$. We then shift $v_1.y$ so that v_1 lies below the line w, New_Peak. Thus, we set $v_1.y = \beta * (v_1.y - w.y) + w.y$, for any $0 < \beta < 1$. Note that, because of our shifting and rotating of the axis, and the moving of v to New_Peak, any value for β between 0 and 1 is correct, say $\beta = 0.9$. Now, to determine the y-coordinate of v_i, $i \geq 2$, we compute $v_{i-1}.y$ plus a fraction α ($\alpha < 1$) of $v_{i-1}.y - v_{i-2}.y$ ($i \geq 3$). Then, the resulting points will be suitably placed. Figure 3.32 shows this idea. We refer to the difference between y-coordinates of successive apexes as a y-increment. Since the y-increments are always decreasing, if we ensure that the initial y increment $v_1.y - v_0.y$ is less than the gradient of the line between New_Peak and w, then the points will all lie in the triangle v, w, z, and the resulting polygon will, provided we place the second half of the vertices correctly, be convex.

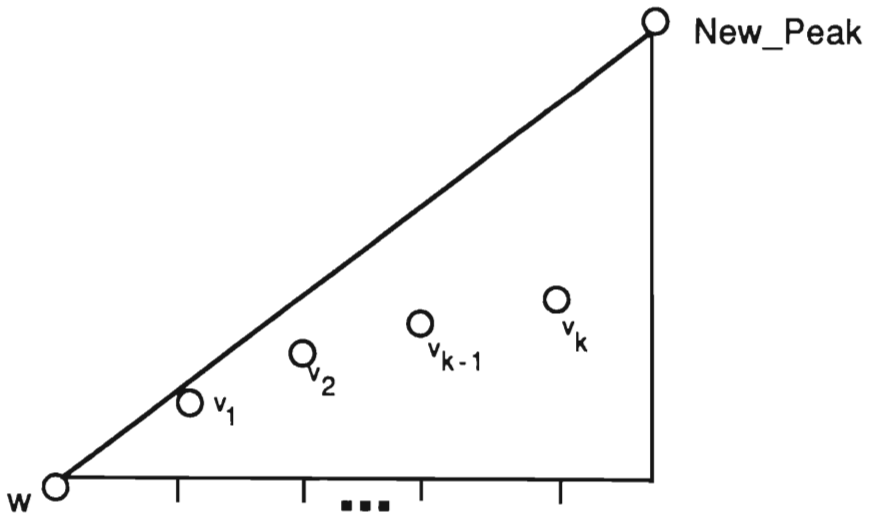


Figure 3.32 - Placement of First Half of the vertices to place

We may now consider the algorithm for placing the first half of the new apexes.

Algorithm 3.18: Place_Half_Vertices (Cycle_Elements)

```
{ set half the vertices y-coordinate in a convex
      manner in the triangle v, w, z }

Compute the gradient m and intercept c of the line w, New_Peak
β = 0.9          { aesthetic coefficient < 1 }
v1. y = β * (v1. y - w. y) + w. y  { place vertex v1 }

Initial_Increment = (v1. y - w. y).
α = 0.9          { aesthetic coefficient < 1 }
Increment = Initial_Increment

For Loop = 2 to |Cycle_Elements| div 2 do
                                   { place the rest of the vertices }
    Increment = Increment * α
    vLoop. y = v Loop-1. y + Increment

If |Cycle_Elements| mod 2 ≠ 0    { special case midpoint }
    then
        Loop = |Cycle_Elements| div 2 + 1
        vLoop. y = v Loop-1. y + Increment * α
```

end

Note that once we have drawn the first $\left\lfloor \frac{n}{2} \right\rfloor$ of the apexes, then if there are an odd number of vertices in Apexes, we draw the special case of a midpoint vertex by merely iterating one more time.

The last $\left\lfloor \frac{n}{2} \right\rfloor$ of the apexes in Apexes may be placed in a similar fashion.

Suppose that the last $\left\lfloor \frac{n}{2} \right\rfloor$ of the vertices to be placed is $v_t, v_{t+1}, \ldots, v_n$.

Once again, we are able to easily compute the initial increment from z to v_n, merely compute the gradient m and intercept c of the line New_Peak, z, and, for some β, assign $v_{2k}. y = \beta * (m * v_n. x + c)$. Now, given the initial increment of $v_n. y - z. y$, we calculate a suitable factor, γ say, so that

$v_{i-1}.y = v_i.y + \gamma^{n-i} *$ Initial_Increment $(t+1 \leq i \leq n-1)$ and also $v_t.y = v_k.y$ (i.e. the two halves of the polygon meet at the same height above the x-axis). Note that the restriction of $v_t.y = v_k.y$ is important. If $v_t.y$ is not equal to $v_k.y$, then we can no longer guarantee that the interior angle of the polygon is less than or equal to 180°.

The value of γ may be determined by using a numerical technique like the Bisection Method (see for example Johnson and Riess [JR82]), which we now describe. The Bisection Method solves functions of the form $f(x) = 0$. We require two starting points a and b, such that $f(a) * f(b) < 0$ (i.e. we have two points to either side of a root), and we require that the function is continuous in the interval [a, b].
Now, we know the desired total y increment, called Total_y_inc, and the starting y increment, called Start_y_inc. Thus we have, from a simple geometric series formula, the following equation

$$\text{Total\_y\_inc} = \frac{(1 - \gamma^k)}{(1 - \gamma)} * \text{Start\_y\_inc.} \qquad (1)$$

or

$$f(\gamma) = \text{Total\_y\_inc} - \frac{(1 - \gamma^k)}{(1 - \gamma)} * \text{Start\_y\_inc} \qquad (2)$$

For some $\gamma = 0.99$, say, $f(\gamma)$ is less than zero, since Total_y_inc is less than New_Peak.y - z.y. Also, for $\gamma = 0$, $f(\gamma) > 0$, by our choice of Start_y_inc. The Bisection Method then proceeds by successively halving the interval [a, b]. Suppose that $c = \frac{(a+b)}{2}$. We choose as new endpoints for the next iteration, the point c, and either a or b, depending on whether a or c lie on the same side of the root or not.

Once γ is determined, we follow Algorithm 3.18 exactly to determine the placement of the second half of the vertices.

Theorem 3.7: Algorithm 3.16 takes linear time and space to complete a convex drawing of G.
Proof: As observed already, each edge is traversed twice, once for each region of G. From Algorithm 3.18, we see that the drawing algorithm takes no more than |Cycle_Elements| to complete. Note that the

numerical iteration routine will converge after a finite number of steps. The Bisection Method converges rapidly to the correct solution. Suppose that our original endpoints of the interval were a and b. After n iterations, the maximum error is $\frac{|a - b|}{2^{n+1}}$, (see for example [JR82]). Thus, the result follows. ❑

The drawing algorithm is conceptually easy to follow. The only information required before the start of the algorithm is an extendible convex polygon S^* of an outer facial cycle S of G, and the corresponding (rotational) embedding of G. In the next subsection, we will present a linear time algorithm for finding an extendible convex cycle S^* of S.

Testing for and Finding an extendible Convex Cycle

A naive approach to finding an extendible convex cycle might be to repeatedly check all facial cycles among all possible embeddings of G in the plane until we find one which satisfies Condition 1. Since this may require us to check an exponential number of facial cycles (since there may be a number of embeddings), this approach is impractical. Chiba, Yamanouchi and Nishizeki [CYN84] use a different approach to the problem, and modify Condition 1 into a form more suitable for an efficient implementation. We shall follow the approach of Chiba and Nishizeki [CN88] for details. Note that we may relax the restriction of no vertices of degree 2 for this section. Thus, we deal with a 2-connected planar graph G. To test if G has an extendible convex cycle, we first need some preliminary definitions.

Suppose that {x, y} is a separation pair of a graph G, and that G is split at {x, y}. The split graphs are then also repeatedly split at {x, y}. The components constructed in this way are called the *{x, y}-split components.* Note that the {x, y}-split components are not necessarily the triconnected components. Figure 3.33, below, gives the {2,3}-split components for a graph G.

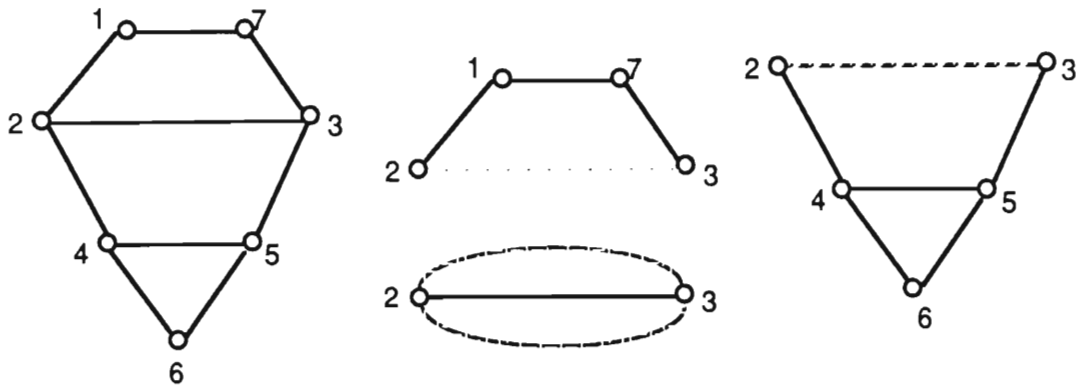


Figure 3.33 - A graph G and the {2, 3} - split components of G.

For the following discussion we refer the reader to the discussion prior to Lemma 3.6 in Section 3.2. We say that a separation pair {x, y} is *prime* if x and y are end vertices of a virtual edge of some triconnected component. We may think of prime separation pairs {x, y} as separation pairs which must be considered when attempting to obtain a convex drawing of a plane graph G. In the above example, the separation pairs {1, 3} or {2, 7} are not prime, since, when we merge split components together, we discard those pairs.

We can now classify the prime separation pairs into three categories, which identify which separation pairs {x, y} cause problems, and which separation pairs may be safely ignored. Firstly, there are the *forbidden* separation pairs. These are prime separation pairs which have either at least four {x, y}-split components or which have three {x, y}-split components, none of which is a ring or a bond. As will be discussed later, forbidden separation pairs automatically imply that G has not got a convex drawing.

The second category are the *critical* separation pairs. As the name suggests, careful consideration must be given to these pairs, if an extendible convex cycle is to be selected. If {x, y} is a critical separation pair, then {x, y} has either three {x, y}-split components, including a ring or a bond, or two {x, y}-split components, none of which is a ring. The last category of separation pairs is implicitly defined by not belonging to either of the above two categories. We say that if {x, y} is separation pair which is

prime, but neither critical nor forbidden, then {x, y} is an *ordinary* separation pair. We may think of {x, y} as being accommodating.

We may now present an important result, a new condition equivalent to Condition 1, which is more suitable for efficient implementation than Condition 1.

Lemma 3.22: Let G be a 2-connected plane graph with outer facial cycle S. Let S^* be a strict convex polygon of S. Then, S^* is extendible if and only if G satisfies the following criteria.

Condition 2

(a) G has no forbidden separation pair

(b) For each critical separation pair {x, y} of G, there exists at most one {x, y}-split component having no edge of S. Also, such an {x, y}-split component is either a bond if (x, y) $\in$ E(G), or otherwise the {x, y}-split component is a ring.

Proof: We shall show, under the restriction that S^* is strict, that Condition 1 implies Condition 2 and vice versa.

Condition 1 implies Condition 2: Let {x, y} be a prime separation pair of G, and let $H_1, H_2, ..., H_k$ be the {x, y}-split components having no edge in S. We shall show that, under Condition 1, either k = 0 or k = 1, and that if k = 1, we have H_1 is either a ring or a bond.

Suppose that such a {x, y}-split component, say H_1, is neither a ring, nor a bond. Then, H_1 contains a vertex v with degree at least 3. However, then Condition 1(a) is contradicted, since all paths from v to S must pass through either vertex x or vertex y. Thus, every H_i must be either a ring or a bond. Next, suppose that k > 1. Then, H_1 and H_2 form a cycle C, which contains no edge of S, and thus, since x and y are the only vertices on C which have degree at least 3, we obtain a contradiction to Condition 1(c).

Since at most two {x, y}-split components contain edges of S, there are at most three {x, y}-split components. Furthermore, one of them must be a ring or a bond. Thus, {x, y} is not a forbidden pair. Suppose now that {x, y} is critical, and that k = 1. If the edge (x, y) $\notin$ E(G), then H_1 is a ring. If the edge (x, y) $\in$ E(G), and (x, y) $\in$ E(S), then the only vertices of the (x, y)-path

in H_1 which are adjacent to vertices of S are those adjacent to x and y. Hence H_1 is a bond, otherwise we have a contradiction of Condition 1(b), since (x, y) is a side of S^*.

Condition 2 implies Condition 1: First, suppose that G has a vertex v of degree at least 3, in V(G) - V(S) which does not satisfy Condition 1(a). Then, using Menger's Theorem (for example, Behzad, Chartrand and Lesniak-Foster [BCL79]) there are two vertices x and y on S which form a separation pair. Since v has degree at least 3, {x, y} is a prime separation pair, and v belongs to an {x, y}-split component, H_i, say. Now, since v is not on S, it follows that H_i contains no edges from S. Lastly, {x, y} is a critical separation pair, since {x, y} is a prime separation pair, and, since H_i contains no edges from S, there are either at least two other {x, y}-split components, or there is one other {x, y}-split component, which is neither a ring, nor a bond. Thus, {x, y} is a critical separation pair, and Condition 2(b) is contradicted. So, Condition 1(a) holds.

Suppose that G does not satisfy Condition 1(b). Then, there exists a component C, such that there are only the two vertices x and y on S adjacent to vertices of C, and $(x, y) \in E(S)$ (since S^* is strict). Therefore, {x, y} is a prime separation pair. Now, we note that the {x, y}-split component containing C has no edge in common with S and cannot, since G is simple, be a bond. Thus, we obtain a contradiction of Condition 2(b).

Lastly, suppose that a cycle C in G which has no edge in common with S does not satisfy Condition 1(c). Since G is 2-connected, there are exactly two vertices, x and y, on C with degree at least 3. Note that {x, y} is certainly a prime separation pair. Now, if $(x, y) \in E(S)$, then an {x, y}-split component having no edge in S is a ring. If $(x, y) \notin E(S)$, then there are two {x, y}-split components having no edge in S. In either case we contradict Condition 2(b). ❑

Note that the restriction of S^* to be strict does not restrict the generality of Condition 2. We have the following lemma that follows directly from Condition 2.

Lemma 3.23: Let G be a 2-connected plane graph with outer facial cycle S. If S has an extendible convex polygon S', then every strict convex polygon S^* of S is also extendible.

Proof: The proof follows directly from Condition 1. By modifying the list of apexes of S' to be strict, we certainly do not contradict Condition 1(a) or 1(c). Since we may reduce, but never increase,the order of each side of S', Condition 1(b) is not contradicted either. Thus, every convex polygon S^* modified in this manner is extendible as well. ❑

We may obtain the following result directly from Lemma 3.23 and Lemma 3.22.

Theorem 3.8: Let G be a 2-connected plane graph with outer facial cycle S, and let S^* be a strict convex polygon of S. Then S^* is extendible if and only if Condition 2 holds.

Thus, we may use Condition 2 to test if G has a convex drawing. We have already seen triconnectivity algorithms in Section 3.2 and Section 3.3, that provide enough information about the separation pairs and the triconnected components of G to test a cycle S for Condition 2. All that remains is a suitable technique for efficiently finding an extendible facial cycle S. We start with a number of observations.

Corollary 3.1: A 2-connected planar graph G has no convex drawing if G has a forbidden separation pair.

Corollary 3.2: A 2-connected planar graph G has a convex drawing if it has no critical or forbidden separation pairs.

From Corollary 3.2, we may immediately obtain the following lemma.

Corollary 3.3: Every 3-connected plane graph G has a convex drawing.

So, from Corollary 3.3, we may obtain the following result which suggests a naive drawing algorithm. Note that a similar result was proved by Fáry [Far46].

Corollary 3.4: Every planar graph G can be embedded in the plane so that each edge is a straight line.

Proof: The result is immediate if $p \leq 3$. Assume thus that $p \geq 4$. Add edges into G to obtain a triangulated planar graph G' (i.e. every region of a planar embedding of G' is a triangle). We now prove that G' is 3-connected. Suppose that {u, v} is a separation pair of G'. Therefore, G' - {u, v} has at least two components H_1 and H_2.

Suppose that u and v do not share a common region of G'. Let $u_1, u_2, \ldots, u_n$ be the vertices adjacent to u, as they appear in anticlockwise order around u in some embedding of G'. Observe that, since G' is a triangulated graph, u_i is adjacent to u_{i+1} $(1 \leq i \leq n-1)$, and that u_n is adjacent to u_1. Hence, there is a cycle $C : u_1, u_2, \ldots, u_n, u_1$ in G'. Now, consider any x - y path P (x, $y \neq u, v$) in G'. If u is on P, then so is u_i and u_j, for some $1 \leq i, j \leq n$. Replace the subpath u_i, u, u_j of P with the part of cycle C from u_i to u_j, to form a new path P' from x to y. We repeat the operation if v is on P', to obtain a path P" from x to y which does not contain u or v. This contradicts the hypothesis, since then {u, v} is not a separation pair.

Suppose now that u and v share a common region with a vertex x. In G' - {u, v}, x belongs to some component H_1. Let the vertices adjacent to u, as we proceed in anticlockwise direction around u, be $x = x_1, x_2, \ldots, x_n$. Suppose that x_k is the first vertex in this sequence which does not belong to H_1. Since G' is a triangulated graph, x_k and x_{k-1} are adjacent, and we obtain a contradiction. So therefore G' is 3-connected.

Now, by Corollary 3.3, G' can be embedded in the plane so that each edge is a straight line. Deleting the edges of E(G') - E(G) from the drawing produces the desired result. ❑

We make an important observation in the following lemma.

Lemma 3.24: If a facial cycle S of a 2-connected planar graph G satisfies Condition 2, then S contains every vertex of the critical separation pairs of G.

Proof: Assume that {x, y} is a critical separation pair of G, and that S does not contain the vertex x. Then, exactly one {x, y}-split component contains all the edges of S. But, Condition 2 implies that exactly one {x, y}-split component contains no edges of S, and that {x, y}-split component must be

either a ring or a bond. But, then {x, y} is not a critical separation pair, contrary to our assumption. ❑

Lemma 3.24 is an important lemma. We determine later those graphs G for which the converse is true to a certain extent. This implies that, to discover an extendible cycle S, we must look at the facial cycles having all the vertices of the critical separation pair vertices. The next lemma simplifies the case where we only have one critical separation pair.

Lemma 3.25: If a 2-connected plane graph G has no forbidden pair, and has exactly one critical separation pair, then G has a convex drawing.
Proof: Let {x, y} be the critical separation pair. Consider Figure 3.33, below. The shaded part of the diagrams represent the {x, y}-split components which are neither a ring nor a bond. It is easy to see that the diagrams in Figure 3.34 represent all the possible cases. Immediately, one may observe that the outer cycle satisfies Condition 2. ❑

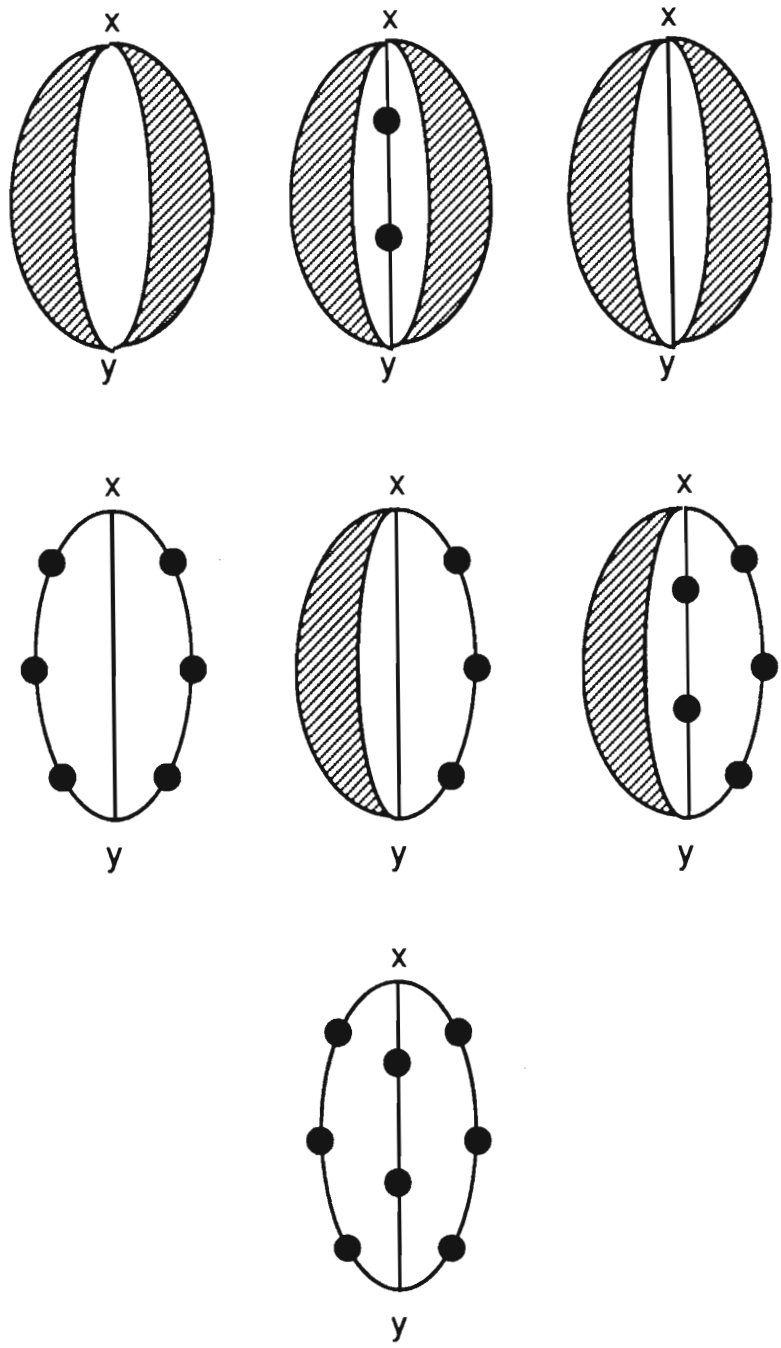


Figure 3.34 - The possible configurations if G has one critical separation pair

Thus, we may now concentrate on graphs with two or more critical pairs. We make one more observation before we present the main result for finding an extendible cycle.

Lemma 3.26: Let G be a 2-connected planar graph, and let {x, y} be a critical separation pair of G. If a facial cycle S contains the vertices x and y, then exactly two {x, y}-split components contain edges of S.
Proof: Since S is a cycle, at most two {x, y}-split components contain edges of S. Also, since S is a facial cycle, and from Lemma 3.24, not all edges of S belong to a single {x, y}-split component. Thus, exactly two {x, y}-split components contain edges of S. ❑

We are now ready to present the main theorem which we use for the testing algorithm.

Theorem 3.9: Suppose that a 2-connected planar graph G has no forbidden separation pairs, and has at least two critical separation pairs. Apply the following operation to G for every critical separation pair {x, y}. If $(x, y) \in E(G)$, then delete the edge (x, y) from G, otherwise, ($(x, y) \notin E(G)$) if exactly one {x, y}-split component is a ring, then delete the x-y path in that {x, y}-split component from G. Let the resulting graph be called G_1.
Then, S is an extendible facial cycle of G if and only if S is a facial cycle of G_1, which contains all the vertices of the critical separation pairs of G.
Proof: Assume that S is the extendible facial cycle of a plane graph G. Then, S satisfies Condition 2. Suppose {x, y} is a critical separation pair. From Condition 2, if $(x, y) \in E(G)$, then $(x, y) \notin E(S)$, and if there is a {x, y}-split component which is a ring, and $(x, y) \notin E(S)$, then that {x, y}-split component has no edges of S. Thus, all the deleted edges and paths are not on S, and hence S remains the outer facial cycle of the plane subgraph G_1 of G. From Lemma 3.24, S contains all the vertices of the critical separation pairs of G.
Assume next that S is a facial cycle of G_1 which contains all the vertices of the critical separation pairs of G. We shall show that every critical separation pair {x, y} of G satisfies Condition 2(b).
By Lemma 3.26, exactly two {x, y}-split components, say H_1 and H_2 of G, contain edges of S. Therefore, since G has no forbidden separation pairs, G has at most one {x, y}-split component not containing edges of S. Suppose that there exists such an {x, y}-split component, H_3 say, and that H_3 is neither a ring nor a bond. Thus H_3 contains no vertex of a critical

separation pair other than x and y, since if H_3 contained a separation pair vertex w ($\neq$ x, y), then H_3 would have to, by assumption, contain an edge of S. Therefore, because G contains at least two critical separation pairs, there is a separation pair {u, w} different from {x, y}, such that neither vertex u, nor vertex w is contained in H_3 - {x, y}. Hence, an {x, y}-split component, H_1 say, is neither a ring nor a bond. Then, H_2 is a ring or a bond. Suppose that H_2 was a bond. But, then the graph edge of H_2 should have been deleted to construct G_1, so H_2 could not contain any edge of S, contrary to assumption. Since, by our assumption, H_3 is neither a ring nor a bond, H_2 is the only {x, y}-split component which is a ring. Hence, the edges of H_2 would have been deleted from G to obtain G_1. This is not possible since H_2 contains edges of S. Thus, our original assumption that H_3 is neither a ring nor a bond is false. Lastly, it is easy to see from the proof that S is a facial cycle of G as well. Consider an embedding of G_1 with S the outer facial cycle. Now, to obtain an embedding of G we may freely place the deleted paths and edges of G inside the facial cycle S, since the endpoints of the paths and edges correspond to separation pairs {x, y}. Thus, S is still an outer facial cycle of an embedding of G.

Hence, S with G satisfies Condition 2(b). ❑

As an example graph G, and corresponding graph G_1 as constructed in Theorem 3.9 is shown in Figure 3.35, below.

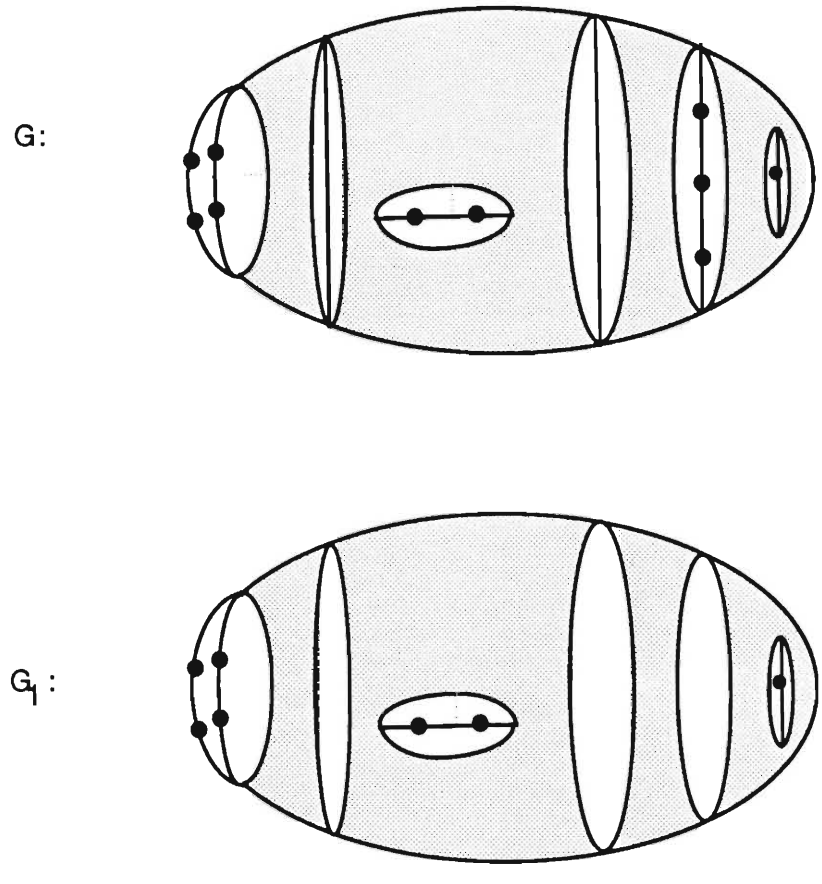


Figure 3.35 - Graph G, and corresponding graph G_1

We need one more corollary before considering the last two results. Let v be a vertex of a 2-connected plane graph G_2. Let $G_1 = G_2 - v$ be 2-connected. Then, the *v-cycle* of G_2 is the facial cycle of the plane subgraph G_1 of G_2 which contains all the vertices adjacent to v in G_2. The following two corollaries use Theorem 3.9 and the concept of a v-block to obtain an algorithm for testing for and finding an extendible convex cycle.

Corollary 3.5: Suppose that a 2-connected planar graph G has no forbidden separation pairs, and at least two critical pairs. Let G_1 be as defined in Theorem 3.9. Construct a graph G_2 from G_1 as follows. Add a new vertex v to G_1, and join every vertex of the critical separation pairs of G to v.

Then, a cycle S is an extendible facial cycle of G if and only if

(a) G_2 is planar, and
(b) S is the v-cycle of a plane embedding of G_2.

The graph G_2 for the example graph G shown in Figure 3.35 is shown below, in Figure 3.36. Note the ease at which we may now obtain an extendible cycle S of G. We merely delete the necessary edges and paths to form G_1, connect the vertices of the critical separation pairs of G in G_1 to a new vertex v, and test for planarity. The following result summarises the results to provide a test for the existence of a convex drawing of a given 2-connected planar graph.

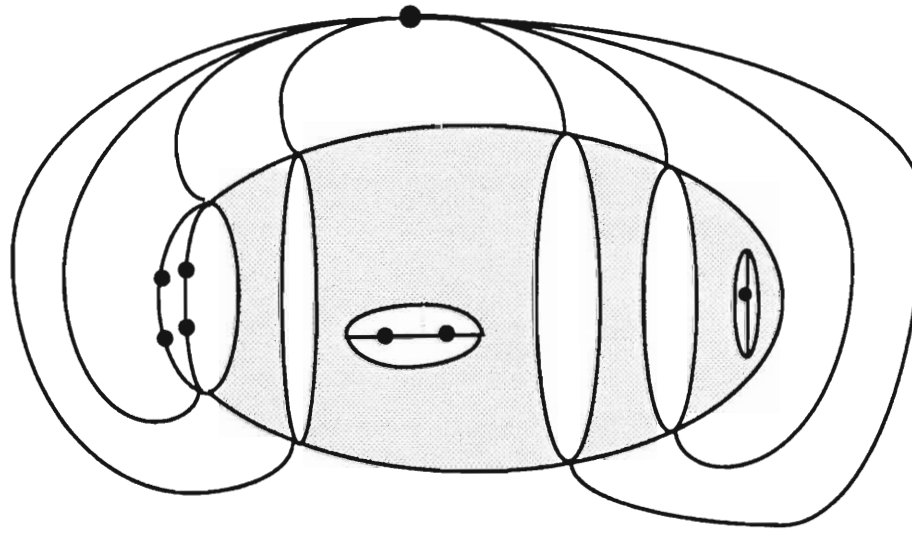


Figure 3.36 - graph G_2

Corollary 3.6: Suppose that a 2-connected planar graph G has no forbidden separation pair, and has two or more critical separation pairs. Let G_2 be the graph defined in Corollary 3.5. The graph G has a convex drawing if and only if G_2 is planar.

Now, using Corollaries 3.1, 3.2, 3.5 and 3.6, and Lemma 3.25, we may obtain the following algorithm.

Algorithm 3.19: Convex_Test_and_Find (G, S)
{ test if a planar graph G has a convex cycle S, and if so output S and a corresponding embedding }

```
Obtain an embedding of G from
    Chiba, Nishizeki, Abe and Ozawa (Section 3.1)
Find all separation pairs via either
    Hopcroft and Tarjan (Section 3.2)
or
    Karabeg (Section 3.3)
Determine the set PSP of Prime Separation pairs
Determine the subset FSP ⊆ PSP of Forbidden Separation pairs
Determine the subset CSP ⊆ PSP of Critical Separation pairs

if |FSP| ≠ ∅
    then
        Output "Forbidden Separation pair detected"
        Output "No cycles are extendible"
        S = ∅
else if |CSP| = ∅
        then
            Output "All cycles are extendible"
            Select S = any facial cycle
        else
            if |CSP| = 1
                then
                    Output "One critical pair"
                    Select the S as in outer cycle in Figure 3.34
                else
                    Construct G1 from G as in Theorem 3.9
                    Obtain G2 from G1 as in Corollary 3.5
                    Test planarity of G2 via
                        Chiba, Nishizeki, Abe and Ozawa (Section 3.1)
                    If G2 is planar
                        then
                            Output "Found a v-cycle"
                            Let S be the v-cycle of G2
                        else
                            Output "No extendible cycles"
                            S = ∅
end
```

Note that the determination of the {x, y}-split components is easy via the existing information in the triconnected components. All that we have to do, is to check that each triconnected component which is a ring does not have more than one virtual edge. If a triconnected component which is a ring has more than one virtual edge, then it must be incident to at least two other triconnected components, none of which is a ring. But then, the ring is no longer a ring in terms of {x, y}-split components. Thus, we

relabel the ring as a 3-connected component (for the purposes of the algorithm only).

The selection of a general facial cycle is relatively easy via the Other_Edge field described earlier in this section. When we categorise the split components into rings, bonds and 3-connected components, we form a list, called *Critical List,* of separation pairs. In the list we note which split components belong to which separation pair. Then, we may determine the sets PSP, CSP and FSP. If |FSP| = ∅, then we delete all ordinary separation pairs from Critical List.

When we attempt to find a facial cycle for the case where |CSP| = 1, we need to know which edges incident with the separation pair vertices x and y belong to which component. Thus, using the information stored when we categorised the split components, we may mark the relevant edges. For example, if we know that we have one 3-connected component and two rings, then we mark the edges in the 3-connected component, and the edges in one of the rings with another mark. Then, we start at vertex x, say, and go through all the facial cycles until we encounter one which includes the edges from both the required components. At worst, we may proceed through O(q) edges, but we only have to find one extendible cycle S.

The v-cycle is found easily by tracing out all the regions which have v as a vertex. Thus, we start at vertex v and proceed to trace out all the regions. At each stage, we merge these regions together, discarding the start and end edges of each region which we have built, hence building the v-cycle.

Note that the service algorithms (i.e. the triconnectivity and embedding routines) called in Algorithm 3.19 have complexity O(p). Thus, from Algorithm 3.19, and the above discussion, we have the following result.

Theorem 3.10: Algorithm 3.19 has complexity O(p).

Chapter 4

Non-Planar Graphs

The preceding chapters have dealt entirely with testing if a (2-connected) graph is planar, or describing embeddings of given planar graphs. In this chapter, we shall look at issues concerning non-planar graphs.

If a graph representing an electronic circuit fails a planarity test of Chapter 2, we may still want to describe a layout for the circuit, but using several layers of a circuit board. This approach is in common use, with the most common circuit boards of this type being *double-layered* circuit boards. The *graph planarization* problem is the problem of partitioning the edge set E(G) of a non-planar graph G into a minimum number of sets $E_1, E_2, ..., E_m$ such that $< E_i >$ is planar for $1 \leq i \leq m$. In Section 4.1 we study an $O(p^2)$ algorithm for finding an approximate solution to the graph planarization problem.

From Chapter 1, we know that a graph G is non-planar if and only if it contains a subdivision of K_5 or $K_{3,3}$. Section 4.2 considers the application of PQ-tree algorithms to the problem of detecting "obstructions" to planarity, i.e. subgraphs which are subdivisions of K_5 or $K_{3,3}$. If a graph is non-planar, it is often important to realise which subgraph of G is an obstruction to planarity. For example, if we model an electronic circuit by a graph G, and we know that a subgraph H is non-planar, we may be able to rework the logic of the electronics of the sub-circuit represented by H. In particular, we study an algorithm by Karabeg [Kar88] which detects and outputs, in linear time, a subgraph of a non-planar graph G, which is a subdivision of the Kuratowski subgraphs K_5 or $K_{3,3}$.

Finally, in Section 4.3, we discuss the problem of finding an upper bound on the genus of a graph. This problem has applications to circuit layouts. By punching holes into the circuit board, one may be able to lay out the circuit so that no two wires cross. The number of holes in the surface equals the genus of the surface. Thus, finding an algorithm which

approximates the genus of the graph representing the circuit is important. We present a new algorithm for finding an upper bound on the genus of a graph.

Section 4.1

The Maximum Planar Subgraph Problem

The problem of finding the thickness of a graph is known to be NP-hard ([Man83]). This suggests trying to find an approximation to the thickness of a graph. In particular, one may wish to take a greedy approach as follows: Begin by finding a planar subgraph G_1 of a non-planar graph G which has a maximum number of edges, and let $G_1' = G - E(G_1)$. Then repeat the process with G_1' to obtain a planar subgraph G_2 which has a maximum number of edges in G_1', and let $G_2' = G - E(G_2)$. The process stops when some G_n' is empty. Thus, the thickness of G is bounded by n. The only difficulty with this approach is that for a non-planar graph G, determining the minimum number of edges, whose removal produces a planar subgraph G, is an NP-complete problem (Garey and Johnson [GJ79]). The problem of finding a planar subgraph of a graph which has a maximum number of edges is called the planar subgraph problem.

A subgraph H of a graph G is a maximal planar subgraph if it is planar and not a proper subgraph of another planar subgraph of G. Thus, instead of developing algorithms for finding planar subgraphs with a maximum number of edges, we develop in this section efficient algorithms for finding maximal planar subgraphs.

A simple algorithm to determine a maximal planar subgraph of a non-planar graph G, is given below.

Algorithm 4.1: Maximal_Planar_Subgraph

{ Determine a maximal planar subgraph of a non-planar graph G }

```
E(H) = ø
For each edge e ∈ E(G)
    E(H) = E(H) + e
    Test_Planarity (H)
        using either
            Hopcroft and Tarjan (Section 2.2)
```

```
            or
                Lempel, Even and Cederbaum (Section 2.3-2.5)
        if H is non-planar
            then
                E(H) = E(H) - e
end;
```

The algorithm is a *brute force* algorithm. It tries every possible edge in G, making no attempt to make intelligent judgements at each stage of the algorithm. The usefulness in Algorithm 4.1 is that it provides an indication on the algorithm complexity of a more efficient algorithm for finding maximal planar subgraphs. We note that Algorithm 4.1 has complexity O(pq), since we try every edge, and test for planarity every time we add an edge. In a worst case, the non-planar graph G is the complete graph, and, in this case, $q = \frac{p(p-1)}{2}$. Thus, an O(pq) algorithm is also $O(p^3)$ for graphs on p vertices and q edges.

The approach used in most algorithms which attempt to solve the maximum planar subgraph problem is to modify the linear time complexity planarity testing algorithms, to delete as few edges as possible to enable the algorithm to continue testing for planarity. Chiba, Nishioka and Shirakawa [CNS79] modified the planarity testing algorithm of Hopcroft and Tarjan (Section 2.2), to produce a maximal planar subgraph algorithm, but their algorithm has complexity O(pq) and uses O(pq) space. Algorithm 4.1 is of the same worst case complexity, so the algorithm by Chiba, Nishioka and Shirakawa is not considered significant in terms of algorithmic complexity. However, Ozawa and Takahashi [OT81] considered modifying the linear implementation of the algorithm by Lempel, Even and Cederbaum [LEC67], to produce a maximal planar subgraph. Their algorithm has an average case complexity of $O(p^k)$, where $k \approx 1.5$. However, the worst case (time) complexity is O(p(p+q)) = O(pq) and requires O(p+q) space. Moreover, Thulasiraman, Jayakumar and Swamy [TJS86] showed that the maximal planar subgraph algorithm by Ozawa and Takahashi does not, in general, produce a maximal planar subgraph when applied to a non-planar graph. The difficulty with the algorithm is that the planar subgraph it produces need not be a maximal planar subgraph. However,

it does produce a maximal planar subgraph when the complete graph K_p is used as input.

The algorithm by Ozawa and Takahashi [OT81] is not without merit. Firstly, they showed that, when their algorithm was applied to random graphs, the actual st-numbering was unimportant. On a graph G with 100 vertices and 500 edges, using different st-numberings the algorithm deleted 332.5 edges on average, with a standard deviation of only 4.3. This fact is useful, in that it simplifies any maximal planar subgraph algorithm using PQ-trees and the vertex addition approach. From a statistical point of view, we do not have to consider the actual st-numbering of the graph, and can concentrate on the reduction and bubble passes. Secondly, Jayakumar, Thulasiraman and Swamy [JTS89] have given an $O(p^2)$ maximal planar subgraph algorithm based on the algorithm by Ozawa and Takahasi.

We shall first discuss the fundamental concepts of the original algorithm by Ozawa and Takahasi. Suppose we have a universal set $\mathcal{U}$, any subset S of $\mathcal{U}$, and a PQ-tree T of the class of PQ-trees over $\mathcal{U}$. The goal is to reduce the PQ-tree with respect to S. Of course, this may not always be possible. In this case, the algorithm determines a subset S_1 of S of smallest cardinality which needs to be deleted from S to perform a successful reduction. Let S' equal S - S_1. During the algorithm there are five possible labels that a node may have with respect to a subset, R say, of S. If X is a node of a PQ-tree T, then we denote by Frontier(X), the leaves of the maximal subtree of T rooted at X.

Label em : A node X is said to have label em (as in empty) if Frontier(X) consists only of non-pertinent leaves with respect to the set R.

Label fu : A node X is said to have label fu (as in full) if Frontier(X) consists only of pertinent leaves from the set R.

Label pa : A node X is said have label pa (as in partial) if there is a PQ-tree T' equivalent to the PQ-tree subtree rooted at X, such that in Frontier(T') all pertinent leaves from

the set R appear either at the left or right end of Frontier(T').

Label mf : A node X is said to have label mf (as in middle full) if there is a PQ-tree T' equivalent to the PQ-tree subtree rooted at X, such that, in Frontier(T'), all pertinent leaves of Frontier(X) from the set R appear consecutively in the middle of Frontier(T'), with at least one non-pertinent leaf appearing at each end of Frontier(T').

Label ot : A node X is said to have label ot if it may have none of the above labellings.

Recall, from Section 2.4, the definitions of full, empty and partial nodes, which we repeat here for convenience. Consider any node X. If none of the descendants of X which are leaves belong to S, we say that X is *empty*. If all of the descendants of X which are leaves belong to S, we say X is *full*, and lastly if some of the descendants of X which are leaves are in S, but some are not, then we say X is *partial*. A distinction should be drawn between the label given here, because it refers to R, and the characteristics that a node has (with respect to S) during the PQ-tree pruned reduction algorithm described in Section 2.5. The basis of this labelling scheme, when applied to graphs, is justified in the following theorem. From now on we assume the graph G has an st-numbering. We let T_k denote the PQ-tree before the reduction for vertex k ($2 \le k \le p-1$), and let S_k denote the pertinent leaves of T_k for the reduction for vertex k ($2 \le k \le p-1$).

Theorem 4.1: A graph G is planar if and only if, for the reduction for vertex k ($2 \le k \le p-1$), Root(T_k, S_k) has one of the labels fu, pa or mf with respect to S_k.

We say that, for a vertex k ($2 \le k \le p-1$), the pertinent root, Root(T_k, S_k), has a *valid labelling with respect to* S_k, when its label is one of fu, pa or mf with respect to S_k. Note that this is not always the case. For example, when a node which is a descendant of the pertinent root, Root(T_k, S_k) has label mf with respect to S_k, then, the label of Root(T, S) is ot. Now, the idea

of the maximal planar subgraph algorithm is to modify the subtree rooted at Root(T_k, S_k), so that we may assign Root(T_k, S_k) a valid label. This modification process is a removal of a minimum number of leaves from T_k, so as to allow a valid labelling of Root(T_k, S_k). If for every vertex k, $2 \leq k \leq p-1$, we are able to label Root(T_k, S_k) with respect to S_k, one of the labels fu, pa or mf, then we may obtain a planar subgraph of G by removing the edges of G corresponding to the leaves deleted during the reduction passes of the reduction process. Note S_k may have been modified in the process if some of its elements were deleted.

A PQ-tree T_k, for some vertex k ($2 \leq k \leq p-1$), is *irreducible* (*reducible*) if we are unable (able) to directly assign Root(T_k, S_k) a valid labelling. Note that T_1 is always reducible, since there is only one leaf. Similarly, the pertinent root of T_p always has label fu. Now, suppose that T_i is irreducible, for some ($2 \leq k \leq p-1$). For a node X in Pruned (T_i, S_i), let Num_Fu, Num_Em, Num_Pa and Num_Mf be the minimum number of descendant leaves which we have to delete from T_i in order to label X with fu, em, pa or mf respectively. We denote these numbers by nf, ne, np and nm, and denote the ordered 4-tuple of these numbers by [nf, ne, np, nm]. If any one of the labels fu, em, pa and mf cannot be attained, then the corresponding number, nf, ne, np or nm, is undefined. Now, as we shall show, X may be labelled with one of the labels fu, em, pa or mf (or, equivalently, the array [nf, ne, np, nm] may be determined) by deciding on the labels of the children. Thus, for example, the number np of leaves which need to be deleted from T_i to assign X the label pa, might be a function of the number of leaves which need to be deleted from T_i to assign a child of X the label fu, say.

As an example of the algorithm, consider the PQ-subtree of some PQ-tree T, shown in Figure 4.1(a), below. Suppose that the root of the PQ-subtree is the pertinent root of our reduction with respect to the set S_k. The pertinent leaves are shaded. Figure 4.1(b) shows the array [nf, ne, np, nm] for every node in the PQ-subtree. So, for example, to assign the Q-node the label pa requires the deletion of one pertinent leaf. To assign a valid labelling to Root(T, S_k), we select the minimum of the set [5, 5, 3, 2], i.e. to assign Root(T, S_k) the label mf requires the deletion of two pertinent

leaves from the frontier of the subtree rooted at Root(T, S_k). Figure 4.1(c) shows the resulting PQ-subtree, with pertinent leaves which are selected for deletion shaded.

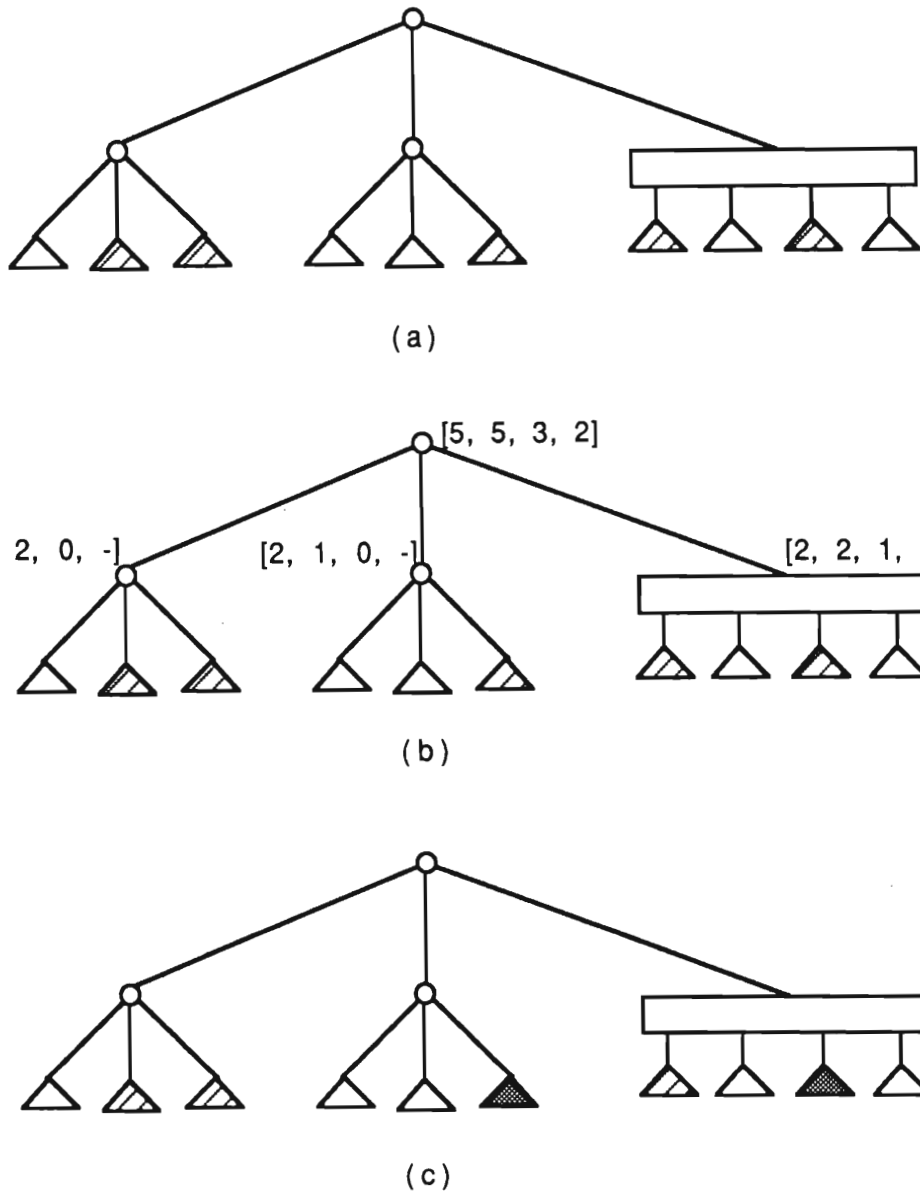


Figure 4.1 - An illustration of the planar subgraph algorithm; (a) a PQ-subtree; (b) after the calculation of 4-tuples; (c) after nodes have been selected for deletion

Now, we traverse T_i from the bottom up, from the leaves of S_i to Root(T_i, S_i) (as in the bubble pass), and determine the array [nf, ne, np, nm] for each node. For a node X, the array [nf, ne, np, nm] may be determined by

looking at the pertinent and non-pertinent children of X. Note that the calculation of the numbers also determines the nodes which are to be deleted. We omit these details, which may be found in the original paper by Ozawa and Takahasi, but an important point, from the algorithmic complexity point of view, is that every node knows its parent at all times.

Once the array [nf, ne, np, nm] is determined for every node in Pruned(T_i, S_i), we may proceed to label every node in Pruned(T_i, S_i). Thus, we may assign Root(T_i, S_i) one of the valid labels fu, pa or mf, by deciding which number in the set {nf, np, nm} is the smallest. After assigning the valid label to Root(T_i, S_i), we may then proceed back down Pruned(T_i, S_i), determining a label for each node.

To assign the label to a node, we merely delete the nodes which are interfering with such a labelling. These nodes were determined during the calculation of the numbers nf, ne, np and nm. Again, we omit these details and refer to the original paper [OT81]. We have the following algorithm giving an outline of the Ozawa and Takahasi algorithm.

Algorithm 4.2: Ozawa_Takahashi_Planar_Subgraph

{ Given a non-planar graph G, the algorithm produces a planar subgraph H by deleting the minimum number of edges at each stage }

```
H = ∅                                   { planar subgraph em }
U = set of edges directed out of v1     { initial universal set }
T1 = Universal tree of U                { the initial tree }
for k = 2 to p - 1 do                   { for each vertex }
    Sk = set of leaves in Tk representing edges directed into vertex vk

    perform bubble pass, Bubble(Tk,Sk),
        and determine array [nf, ne, np, nm] for each node in Tk
    Determine Min [nf, np, nm] for Root(T, S)

    For each node X ∈ Pruned(Tk,Sk), proceeding in a top-down manner
do
        Delete the necessary children,
            and their descendants of X to assign X the correct label

                { add valid edges to H }
    H = H + all edges directed into vk represented by leaves in Tk
```

```
Reduce (T_k,S_k)                                  { do reduction }
S' = set of edges directed out of vertex v_k      { obtain new leaves }
T_{k+1} = Vertex_Addition (T_k, S')               { add new, remove old }
U = (U - S_k) ∪ S'                                { update universal set }
```

end

Notice that we only add the edges to the planar subgraph H which are represented by leaves which are still in the PQ-tree. We have omitted a number of details from the algorithm, the most important of which is the determination of the children nodes to delete at each stage.

As shown by Thulasiraman, Jayakumar and Swamy [TJS86], Algorithm 4.2 may not even produce a spanning planar subgraph. The reason for this is simply because we allow the deletion of both pertinent and non-pertinent nodes from the PQ-tree T. Thus, it may happen that, for a vertex i ($3 \leq i \leq p-1$), all leaves representing edges directed into i were deleted from T_i. Then, vertex i, and possibly some others, are not represented in T_i, and so the planar subgraph produced will not be spanning. We now present an algorithm by Thulasiraman, Jayakumar and Swamy which modifies the above algorithm to delete only pertinent nodes.

Modified Planar Subgraph Algorithm

The following theorem forms the basis of the modified planar subgraph algorithm.

Theorem 4.2: The planar subgraph algorithm, Algorithm 4.2, will determine a spanning planar subgraph H of a 2-connected non-planar graph G, if only pertinent leaves are selected for deletion while making any tree T_k ($2 \leq k \leq p-1$) reducible.

Proof: The proof follows from the fact that a single leaf is reducible. Thus, not all pertinent leaves will be deleted prior to a reduction, and so each vertex will always be connected to a vertex of lower st-number. Hence, H will be spanning. ❑

Suppose that a PQ-tree T_i ($3 \leq i \leq p-1$) is irreducible. In order to make T_i reducible, we must compute the array [nf, ne, np, nm] for every node in Pruned(T_i, S_i). Consider the valid labellings that a node X of Pruned(T_i, S_i) may be assigned, if we may not delete non-pertinent nodes. If X is full, then X, and its descendants, may be labelled em, pa or mf, or they may be labelled fu. However, if X is partial, then it may be labelled em (if all of the pertinent leaves of the subtree rooted at X are deleted), pa (if some or possibly none of the pertinent leaves of the subtree rooted at X are deleted) or mf (again, if some or possibly none of the pertinent leaves of the subtree rooted at X are deleted), but never fu, since we never delete non-pertinent nodes from T_i. Therefore, any partial node in T_i may be labelled em, pa or mf only. Thus, we only need to compute the array [ne, np, nm] for the nodes in Pruned(T_i, S_i).

We now develop the formulas to determine the array [ne, np, nm] for each node X in Pruned(T_i, S_i). As before, we determine the array [ne, np, nm] in a bottom-up manner, starting from the leaves of S_i. When trying to determine the minimum number of leaves which must be deleted to be able to assign a node the label pa, it may happen that this number equals the minimum number of leaves which we must delete to be able to assign a node the label fu. Similarly, when trying to determine the minimum number of leaves which must be deleted to be able to assign a node the label mf, it may happen that this number equals the minimum number of leaves which we must delete to be able to assign a node the label pa or fu.

We denote the numbers ne, np, nm for a pertinent child j of X by the numbers ne_j, np_j, nm_j. Let Pert(X) denote the pertinent children of X, and let Par(X) denote the children of X with label pa. To determine the array [ne, np, nm] for X, we have the following four cases, depending on what type of node X is.

Case 1: X is a pertinent leaf. In this case, ne = 1, np and nm are zero.

Case 2: X is a full node. Then, np and nm are zero. To assign X the label em, we must delete every pertinent child of X, thus

$$ne = \sum_{j \in Pert(X)} ne_j.$$

We delete the maximal subtree rooted at X.

Case 3: X is a partial P-node. Again, to assign X the label em, we must delete every pertinent child of X, thus

$$ne = \sum_{j \in Pert(X)} ne_j$$

To assign X the label pa, we may make all full children have label fu, assign to one of the partial children the label pa, and to all other partial children the label em. Thus, the value of np is given by

$$np = \left(\sum_{j \in Par(X)} ne_j \right) - \max_{j \in Par(X)} \{(ne_j - np_j)\}$$

We may assign X the label mf in two ways. Firstly, we may assign the one partial child the label mf, and all other pertinent children the label em. In this case, the number of leaves to delete is given by

$$\alpha_1 = \sum_{j \in Pert(X)} ne_j - \max_{j \in Par(X)} \{(ne_j - nm_j)\}$$

Otherwise, we may assign two partial children the label pa, all full children the label fu, and all other pertinent children the label em. In this case, we get

$$\alpha_2 = \sum_{j \in Par(X)} ne_j - \max1_{j \in Par(X)} \{(ne_j - np_j)\} - \max2_{j \in Par(X)} \{(ne_j - np_j)\}$$

where max1 and max2 return the the maximum and the second maximum respectively. Thus, a P-node X may be assigned the label mf by deleting

$$nm = \min \{\alpha_1, \alpha_2\}$$

pertinent nodes from T_i.

Case 4: X is a partial Q-node. To assign X the label em, we merely delete all pertinent children of X, thus

$$ne = \sum_{j \in Pert(X)} ne_j.$$

To assign X the label pa needs a little more work. Firstly, note that X may only be assigned the label pa if at least one of its endmost children are pertinent (since we are only deleting pertinent nodes). Denote by $Pert_L(X)$ and $Pert_R(X)$, the maximal consecutive sequence of pertinent children, starting from the left and right endmost child respectively, so that only the last (i.e. the rightmost

and leftmost respectively) child in the sequence may be partial. If the left (or right) endmost child is not pertinent, then $P_L(X)$ ($P_R(X)$) is empty. If both $P_L(X)$ and $P_R(X)$ are empty, the value of np is undefined. Suppose now that at least one of $P_L(X)$ or $P_R(R)$ is non-empty. Given these two lists, we may assign X the label pa by deleting

$$np = \sum_{j \in Pert(X)} ne_j - \max \left\{ \sum_{j \in P_L(X)} (ne_j - np_j) , \sum_{j \in P_R(X)} (ne_j - np_j) \right\}$$

pertinent leaves from T_i.

We may assign the label mf to X in two different ways. Firstly, we may assign one of the children of X the label mf, and assign all other children the label em. This may be achieved by deleting

$$\beta_1 = \sum_{j \in Pert(X)} ne_j - \max_{j \in Pert(X)} \{(ne_j - nm_j)\}$$

pertinent leaves from T_i. Now, let $P_A(X)$ be a maximal consecutive sequence of pertinent children of X, such that, every child in the sequence must be full, except possibly the starting or ending children of the sequence, which may be full or partial. We may assign X the label mf by assigning all full children in $P_A(X)$ the label fu, all partial children in $P_A(X)$ the label pa, and all other pertinent children the label em. Thus, we may delete

$$\beta_2 = \sum_{j \in Pert(X)} ne_j - \max_{\forall P_A(X)} \left\{ \sum_{j \in P_A(X)} (ne_j - np_j) \right\}$$

pertinent leaves from T_i. Therefore, the minimum number of pertinent children which we must delete from X in order to assign X the label mf is

$$nm = \min \{\beta_1, \beta_2\}$$

It is easy to see that these numbers can be easily computed during the bubble pass of the pruned reduction algorithm by looking only at pertinent children of each node in Pruned(T_i, S_i). We call this stage of the planar subgraph algorithm, the *compute numbers algorithm*.

In Cases 3 and 4, we need to note which partial children were not selected by the procedure for deletion. Consider Case 3 (i.e. X is a P-node). To compute np for X, let partial1(X) be the partial child assigned the label pa.

When we compute nm, we have two cases. If nm = α_1, then let medfull(X) be the partial child assigned the label mf. On the other hand, if nm = α_2, then let partial1(X) and partial2(X) be the two partial children, selected by max1 and max2, respectively, and we let medfull(X) be empty. Consider Case 4 (i.e. when X is a Q-node). When computing the number np, we let partial1(X) be the rightmost node or leftmost node of the sequence $P_L(X)$ or $P_R(X)$, depending on which sequence was selected in the computation of np. When calculating the number nm if nm = β_1, we let medfull(X) denote the partial child of X to which we assign the label mf. If nm = β_2, we denote by partial1(X), an endmost child in the sequence $P_A(X)$, and we let medfull(X) be empty.

Lemma 4.1: The compute numbers algorithm correctly computes the array [ne, np, nm], for all pertinent nodes, during all the passes of the pruned reduction algorithm, and its overall time complexity is $O(p^2)$.

Proof: The correctness of the algorithm follows directly from the discussion preceding this lemma. To determine the complexity of the compute numbers algorithm we consider two cases, depending on the type of any pertinent node X in a PQ-tree T_i.

Case 1: Suppose X is a Q-node. The algorithm compute numbers traverses all the children of X exactly once. Thus, the amount of work performed for all the Q-nodes in T_i is proportional to the number of children of all the Q-nodes in T_i. Note that a child of X corresponds to a vertex on the boundary of the outer region of the 2-connected subgraph represented by X. Also, every vertex is represented (if at all) by a child of a most one Q-node. Thus, we may bound the total number of Q-nodes in T_i to p.

Case 2: Suppose X is a P-node. We associate, with X, a list of the pertinent children of X. This list is updated in the same manner as the pertinent children field is updated (see Section 2.5). The complexity of the compute numbers algorithm, for X, is proportional to the number of pertinent children of X. Consider any child Y of X. We have three subcases, since Y is either a Q-node, a P-node, or a leaf.

Subcase 2.1: Suppose that Y is a leaf. In this case, the number of pertinent leaves in T_i is merely the in-degree of vertex i.

Subcase 2.2: Suppose that Y is a Q-node. We note that, from Case 1, the number of Q-nodes in T_i is bound by p.
Subcase 2.3: Suppose that Y is a P-node. We observe that the number of P-nodes in T_i is bound by i-1. This may be seen by observing that we introduce P-nodes only during the vertex addition stage of the pruned reduction algorithm, when we replace full nodes and descendants by a new P-node with children being leaves representing edges directed out of vertex i. Any P-node introduced during the reduction process is full, and consequently is removed during the vertex addition stage of the algorithm.
Thus, the complexity of compute numbers for T_i is O(p + in-degree of i).
So overall, for the reductions of trees T_i ($2 \leq i \leq p-1$), the algorithm compute numbers has complexity $O(p^2)$. ❑

After we have computed the array [ne, np, nm] of numbers for each node in Pruned(T_i, S_i), we may then examine the array [ne, np, nm] for Root(T_i, S_i). If the minimum of np and nm is zero, then T_i is reducible for S_i, and we may proceed with the reduction. On the other hand, if the minimum of np and nm is not zero, then T_i is not reducible for S_i. In this last case nf is undefined.

Consider the latter case. We need to assign labels to all pertinent nodes to allow the reduction to proceed. Thus, if T_i is not reducible, then we assign Root(T_i, S_i) the label pa or mf, and proceed in a breadth first manner down the subtree rooted at Root(T_i, S_i), assigning labels to nodes. Note that we may not assign Root(T_i, S_i) the label em, since this would imply that we must delete all pertinent leaves from T_i, which is clearly not the minimum number of leaves which we should delete to make a reduction possible. Now, consider any node X in Pruned(T_i, S_i).
Case 1: Suppose that X has label fu (note $X \neq$ Root(T_i, S_i)). In this case, we would like to keep X and its descendants in T_i, thus, no action must be taken.
Case 2: Suppose that X is not labelled fu. An easy case is when X is a leaf. Hence, X must have the label em, and so we must delete it from T_i. If X is not a leaf, then there are several subcases.

Subcase 2.1: Suppose X has label em. In this case we must assign the label em to each of the pertinent children. If any of the pertinent children are full, then we must delete the entire subtree of T_i rooted at the full pertinent child.

Subcase 2.2: Suppose X is a P-node, and that X has label pa. In this case, we assign the partial child partial1(X), the label pa, all other partial children the label em, and all full children the label fu.

Subcase 2.3: Suppose X is a Q-node, and that X has label pa. In an analogous manner to the previous subcase, we traverse the children of X from partial1(X) along the sibling chains, and determine the maximal consecutive sequence of children $P_L(X)$ or $P_R(X)$. If the endmost child of $P_L(X)$ or $P_R(X)$, which is not partial1(X), is a partial child, then we label it pa, otherwise we assign it the label fu. All other children in the sequence $P_L(X)$ or $P_R(X)$ are assigned the label fu.

Subcase 2.4: Suppose X is a P-node, and that X has label mf. If medfull(X) is not empty, then we assign the pertinent child medfull(X) the label mf, and all other pertinent children are assigned the label em. Otherwise, if medfull(X) is empty, then we assign both partial1(X) and partial2(X) the label pa, all full children of X the label fu, and all other partial children of X the label em.

Subcase 2.5: Suppose X is a Q-node, and that X has label mf. If medfull(X) is not empty, then we assign the pertinent child medfull(X) the label mf, and all other pertinent children are assigned the label em. Otherwise, if medfull(X) is empty, then we traverse the children of X, starting from partial1(X), and determine the maximal consecutive sequence $P_A(X)$ of pertinent children of X. Then, we assign all interior children in $P_A(X)$ the label fu, the endmost children of $P_A(X)$ the label fu or pa, depending on whether they are full or partial, and all other pertinent children of X the label em.

From the above discussion we obtain the following lemma.

Lemma 4.2: The assignment of label fu, em, pa or mf to a node X of Pruned(T_i, S_i), uniquely determines the labels of its pertinent children.

Lemma 4.2 suggests a top-down processing technique to label the nodes in Pruned(T_i, S_i). We determine the label of Root(T_i, S_i), and then we determine the labels of the descendants of Root(T_i, S_i) in turn. This process of assigning labels to the nodes, and deleting any pertinent nodes with label em, we call the *delete nodes algorithm*. We have the following result.

Lemma 4.3: The delete nodes algorithm, for all T_i ($2 \leq j \leq p-1$), has complexity $O(p^2)$.
Proof: The proof of the complexity follows directly from Lemma 4.1, and by observing that the two algorithms process the same nodes in Pruned(T_i, S_i), for some i ($2 \leq i \leq p-1$). ❑

Once the PQ-tree T_i has been modified to enable the reduction with respect to S_i to proceed, we may reduce T_i. A problem occurs however, with the bubble pass and parent pointers. As was discussed in Section 2.5, in the bubble pass we are sometimes able to detect that a reduction will fail. If, for example, there are two groups of blocked nodes at the end of the bubble procedure, then we may conclude that T_i is irreducible for the set S_i. However, we wish the bubble pass algorithm to continue assigning valid parent pointers to pertinent nodes. The original Booth and Lueker bubble pass algorithm does not allow for this. Thus, we discuss the modifications necessary to enable the bubble pass algorithm to proceed even when T_i is irreducible.

We adopt the convention that, if an interior node has a non-empty parent pointer, then that parent pointer is valid. When we wish to assign a proper parent pointer to a child Y of a Q-node, we traverse the children of the Q-node, starting at Y, until we encounter a child Z with a non-empty parent pointer. At this stage we then assign proper parent pointers to every node which we visited to get to Z from Y. Thus, some form of queuing mechanism is necessary. We add all the children which we traverse when proceeding from Y to Z, to a queue called *interior*. Then, when we reach Z, we must assign the valid parent pointer to every node in the queue. At this stage, if an interior node has a non-empty parent pointer, then that parent pointer is valid. Note, however, that at this stage

there may be several non-pertinent children with non-empty parent pointers. To adhere to the above convention, after the reduction, we need to set all the parent pointers of non-pertinent interior children to empty. This is necessary because the parent pointers may have been invalidated during the reduction (i.e. the parent has been changed). Thus, we keep the queue interior until the end of the bubble pass. Then, we set the parent pointer of every non-pertinent node in interior to empty.

Lemma 4.4: The bubble pass of the planarization algorithm has complexity $O(p^2)$.
Proof: The proof follows directly by observing that there are O(p) children of Q-nodes in T_i. ❑

The last problem with the planar subgraph algorithm is that the two sub-algorithms compute numbers and delete nodes require knowing whether a node is full or partial. This may only be determined by knowing the number of descendant leaves of every node. Thus, we adopt another convention. Every node knows the number of its descendant leaves. We say that a leaf has a single descendant leaf. The rest of the nodes in the PQ-tree must be properly maintained. Thus, every time we add or delete nodes from T_i, we must update these fields. There are three different times during a reduction of T_i with respect to S_i that such modifications are made. Firstly, when we call algorithm delete nodes, secondly during the vertex addition stage of the reduction algorithm, when we remove the pertinent nodes from T_i, and lastly, also in the vertex addition stage, when we add the leaves representing edges directed out of vertex i. During the vertex addition stage of the algorithm, we only need to update the number of descendants of a node once, and the update is by the out-degree of i less in-degree of i. When we call algorithm delete nodes, we also update the descendant leaf count of the ancestors of the nodes. We call this algorithm to update the descendant leaf counts, the *update descendants count* algorithm. We have the following lemma.

Lemma 4.5: The algorithm update descendants count has overall complexity $O(p^2)$ during the total running of the planar subgraph algorithm.

Proof: The lemma follows from Lemma 4.1 by observing that there are O(p) nodes in every T_j ($1 \leq j \leq p-1$). ❑

We may now present the planarize algorithm.

Algorithm 4.3: Jayakumar_Thulasiraman_Swamy_Planar_Subgraph
{ Given a 2-connected non-planar graph G, find a spanning planar subgraph G_p of G.

```
Construct initial PQ-tree T2
for i = 2 to p-1 do
    Bubble_Up (Ti)              { get proper parent pointers etc }
    Compute Numbers (Ti)        { compute number to delete for pa, mf, em }

    if min {np, nm} of Root(Ti, Si) ≠ 0        { see if delete any edges }
        choose min {np, nm} and label Root(Ti, Si) accordingly
        Delete Nodes (Ti)                      { delete edges if so }

    Reduce (Ti)                 { perform reduction }
                                { and do vertex addition stage }
    Remove Full Children and descendants
    Add new leaves being edges directed out of vi
    Update Descendants Count
```

end

Theorem 4.3: Algorithm 4.3 will produce a spanning planar subgraph of a non-planar graph G, and has complexity $O(p^2)$.
Proof: The proof proceeds directly from Lemmas 4.1 through 4.5. ❑

Finding a 2-connected planar subgraph of a non-planar graph

Jayakumar, Thulasiraman and Swamy [JTS89] failed to discuss the properties of the planar subgraphs which they produce. They did, however, note that the subgraphs produced by Algorithm 4.3 may not be maximal, and may not be 2-connected. An important property of the planar subgraph is its connectivity. Later in this section we shall present a maximal planarization algorithm, which, given a 2-connected planar subgraph of a non-planar graph G, will find a maximal planar subgraph of G.

In this sub-section, we show that the vertex addition approach of Jayakumar, Thulasiraman and Swamy does not lead to an easy extension of Algorithm 4.3 to produce 2-connected planar subgraphs, if they exist at all. Consider the non-planar graph G, shown in Figure 4.2, below.

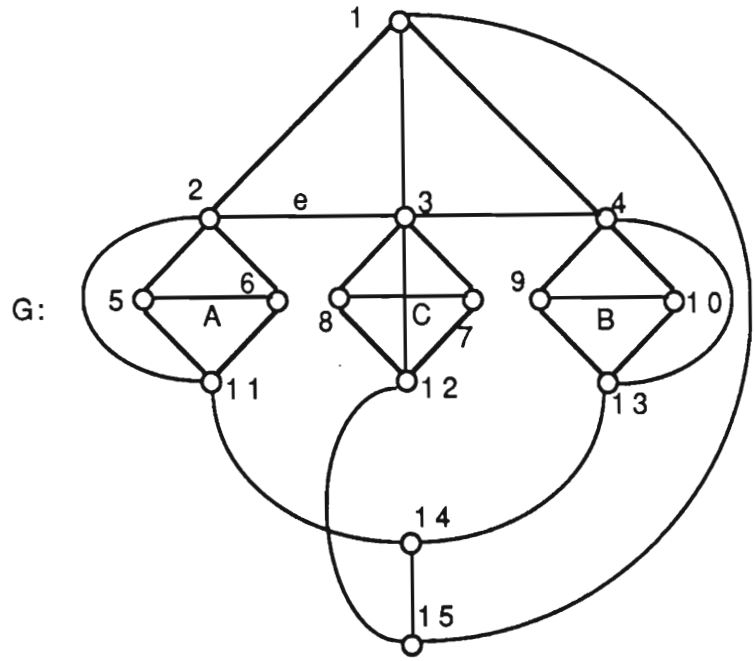


Figure 4.2 - An st-numbered graph G

Now, the reduction process in Algorithm 4.3 for G will proceed very successfully, until we reduce for vertex 14. At this stage we have to delete one of the single edges from the 3-connected components A or B in order to allow the edge from C to vertex p to remain in G. But, as soon as we delete that edge G is no longer 2-connected.

Figure 4.2 illustrates an important flaw in Algorithm 4.3. Using the standard vertex addition approach, it is not sufficient to look only at pertinent leaves. It is not even sufficient to consider both pertinent and non-pertinent leaves at a particular reduction stage. Rather, one must also consider removing edges already accepted into our planar subgraph. In G, if we were to delete the edge e instead of one of the single edges

emanating from the 3-connected components A or B, then we would obtain a 2-connected planar subgraph of G.

The second important question to answer is, does every 2-connected non-planar graph G have a planar 2-connected spanning subgraph G_p? We now answer this open question by Jayakumar, Thulasiraman and Swamy.

Proposition 4.1: Not every 2-connected non-planar graph G has a planar 2-connected spanning subgraph G_p.

Proof: The proof proceeds by construction. Select one of the known Kuratowski graphs K_5 or $K_{3,3}$, choose K_5, say. We construct a new graph G, which is a subdivision of K_5.

See Figure 4.3. Let the vertices of K_5 be $v_1, v_2, ..., v_5$. Now, to construct G, we subdivide each edge of K_5 exactly once, and let the new vertices of V(G) - $V(K_5)$ be $v_6, v_7, ..., v_{15}$. So G has $p = 5 + 10 = 15$, and $q = 20$. Now, observe that to obtain a spanning planar subgraph of K_5, we merely remove one of the edges. However, in G, the removal of any one edge will destroy the 2-connectivity of G. Every vertex v_i, $6 \leq i \leq 15$, is adjacent to exactly two vertices, and every edge $e \in E(G)$ is incident to exactly one vertex v_i, $6 \leq i \leq 15$. Thus, the removal of any edge from G ensures that one v_i, $6 \leq i \leq 15$, is adjacent to only one other vertex v_j, $1 \leq j \leq 5$. Thus, we have that v_j is a cut-vertex of G - {e}. ❑

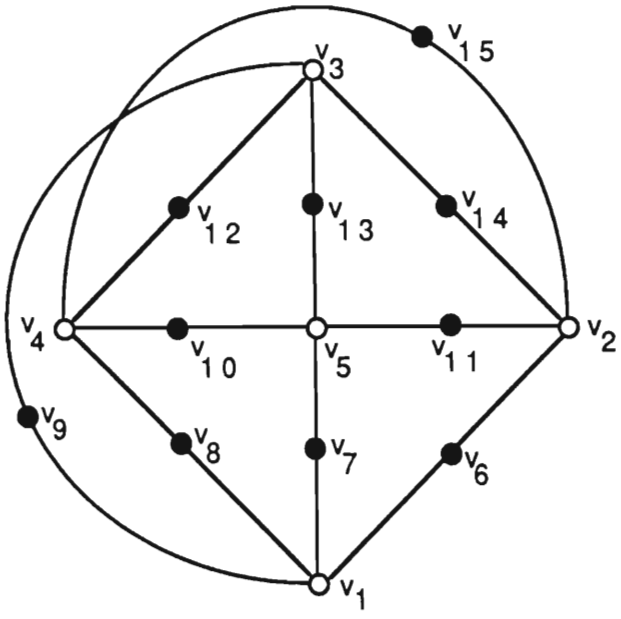


Figure 4.3 - Illustration of Proposition 4.1

One would suspect that a graph with a minimum degree of three might have a spanning 2-connected planar subgraph. However, we have the following result.

Proposition 4.2: Not every 2-connected non-planar graph G, with minimum degree 3, has a planar 2-connected spanning subgraph G_p.

Proof: Again, the proof proceeds by construction. Select one of the known Kuratowski graphs K_5 or $K_{3,3}$, choose K_5, say. We construct a new graph G, which has a subdivision of K_5 as a subgraph as follows.

Let the vertices of K_5 be $v_1, v_2, ..., v_5$. Now, to construct G, we subdivide each edge of K_5 twice, and let the new vertices of $V(G) - V(K_5)$ be $v_6, v_7, ..., v_{25}$. So G has $p = 5 + 20 = 25$, see Figure 4.4. We add the following extra edges into G. Suppose that $e = v_i v_j$, $1 \leq i, j \leq 5$ was an edge of K_5, and we have introduced the two new vertices v_m and v_n, $6 \leq m,n \leq 25$, by subdividing e, such that v_m is now adjacent to v_i, and v_n is adjacent to v_j. Now, we add two extra edges $e_1 = v_n v_i$ and $e_2 = v_m v_j$. Denote the subgraph created by the subdivision of e and the subsequent addition of the edges e_1 and e_2 by $G(v_i, v_j)$. Observe that G now has $q = 50$.

Now, to obtain a spanning planar subgraph of K_5, we merely remove one of the edges. This implies that, in order to obtain a 2-connected planar subgraph of G, we have to delete edges from G to disconnect at least one

subgraph $G(v_i, v_j)$, $1 \leq i, j \leq 5$. Let $G' = G(v_i, v_j)$ be such a subgraph from which we delete edges to disconnect v_i and v_j, and suppose G" is the graph which remains after these edges have been deleted from G'. Then, v_m (as described above) cannot be connected to both v_i and v_j in G". So, suppose v_m is connected only to v_j in G". Then, v_j is a cut-vertex of G" and it follows that no spanning planar subgraph of G is 2-connected. ❑

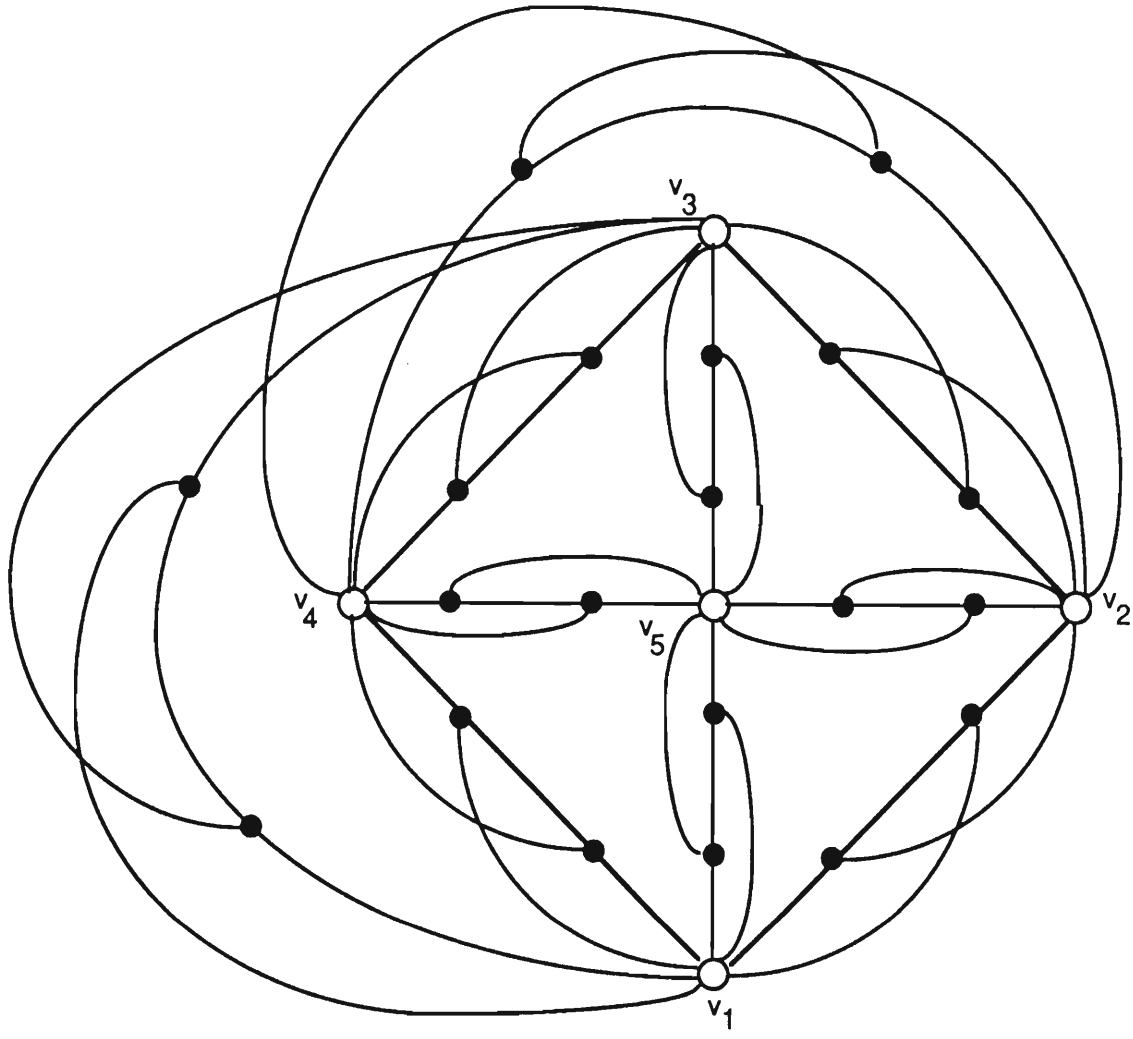


Figure 4.4 - Illustration of Proposition 4.2

The rest of this section is concerned with a maximal planarization algorithm. As input to the algorithm we have a 2-connected non-planar graph G.

Maximal Planarization Algorithm

The key idea of the maximal planarization algorithm, is derived from the following observation. In the planar subgraph algorithm, Algorithm 4.3, during the reduction for a vertex i, say, we may delete a number of pertinent leaves in order to allow the reduction for vertex i to proceed. Some of those leaves need not have been deleted, since the non-pertinent leaves which caused the deletion of these leaves, were themselves deleted at a later stage in the algorithm.

Let the set of edges deleted from G by Algorithm 4.3 during the reduction pass for vertex i be E_i ($5 \leq i \leq p-1$), and let the spanning planar subgraph produced by Algorithm 4.3 be G_p (note E_1, E_2, E_3, E_4 are empty). The maximal planarization algorithm will attempt to add edges from the sets E_i ($5 \leq i \leq p-1$) to G_p, without affecting the planarity of G_p. Note that $\bigcup_{i=5}^{p-1} E_i$ $= E(G) - E(G_p)$.

Now, when calculating the number of leaves to be deleted from T_i to make T_i reducible, we ignore the presence of empty leaves representing edges from $E(G) - E(G_p)$. Let $T_i(G_p)$ denote the subtree of T_i of minimum height whose frontier contains all the pertinent leaves representing edges from G_p. We say that a pertinent leaf representing an edge from G_p is *preferred*. All other pertinent leaves we call *new pertinent leaves* (these leaves correspond to edges from E_i).

Consider the following modified definitions. A node of T_i is called full if its frontier contains no empty leaf from G_p; it is empty if its frontier only has empty leaves; otherwise it is partial. We say that a pertinent node X of T_i is *preferred* if it has some of the preferred leaves in the frontier of the subtree of T_i rooted at X. If X is pertinent but not preferred, then we either retain it in T_i, or delete it, and its descendants, from T_i. With our new definitions of full and partial, we may use the same compute numbers formulae and algorithm as we used in the previous subsection. Consider

now our labelling policy. The following cases describe our modifications to the previous labelling policy.

Case 1: Suppose that X is a P-node, and X is partial. In this case, X can have at most two partial preferred children.

Subcase 1.1: Suppose that X has no partial preferred children. If X is Root($T_i(G_p)$), or an ancestor of Root($T_i(G_p)$), then it may be included in T_i by assigning it the label pa or mf. Otherwise, X is a descendant of Root($T_i(G_p)$), and X may be included only if we assign X the label pa. We set both partial1(X) and medfull(X) to empty.

Subcase 1.2: Suppose that X has exactly one partial preferred child Y. In this case we must retain the partial preferred child, so we set partial1(X) to Y. If X is Root($T_i(G_p)$) or an ancestor of Root($T_i(G_p)$), then we may assign X the label pa or mf. Otherwise, we may only include X by assigning it the label pa. Note that if X is Root($T_i(G_p)$), then none of its children may be assigned the label mf, so medfull(X) will be empty in this case.

Subcase 1.3: Suppose that X has exactly two partial preferred children. Again, we retain both the partial preferred children. We set partial1(X) and partial2(X) to the two partial preferred children. Medfull(X) is set to empty. Lastly, we note that X must be the Root(T_i, S_i) in the reducible T_i.

Case 2: Suppose that X is a partial Q-node. All preferred pertinent children appear consecutively. Thus, we first traverse these children from the leftmost child towards the rightmost child, and determine the maximal consecutive sequence Pref(X), such that

(i) Pref(X) contains all the preferred children;

(ii) only the first or last child in Pref(X) may be partial;

(iii) all other children in Pref(X) must be full.

Note that the above three points are with respect to our new definitions of full and empty. Thus, Pref(X) could have some interior children with empty leaves in their frontier, but the leaves represent edges from the set $E_j \neq E_i$.

To assign X the label pa, the following needs to occur. Pref(X) must appear at the leftmost (or rightmost) side of X, and the leftmost (rightmost) child has no empty leaf representing an edge from G_p. In this case we set $P_L(X)$ (or $P_R(X)$) to Pref(X). We set partial1(X) to the leftmost child of Pref(X).

If X may not be labelled pa, then we have two subcases.
Subcase 2.1: If Pref(X) contains a single child Y, then this child may be assigned the label pa or mf, depending on the corresponding minimum of np and nm, and subject to the condition that it is not assigned mf if Y is a descendant of Root(T_i, S_i). Thus, if Y is assigned the label pa, then we set partial1(X) to Y, otherwise we set medfull(X) to Y.
Subcase 2.2: Suppose that Pref(X) contains more than one child. We set partial1(X) to an endmost child in the sequence, and assign X the label mf. Again, we note that X must be the pertinent root.

If we process the pertinent nodes of T_i up to Root(T_i, S_i), using the compute numbers formulae and algorithm, as modified above, then we can determine the array [ne, np, nm] for every node in Pruned(T_i, S_i). We refer to this modified algorithm as the *modified compute numbers algorithm*.

It is worthwhile noting that the sequence Pref(X) may contain empty (but non-preferred) leaves. It is at this stage that the maximal planarity of the resulting subgraph is ensured. We ignore the presence of such leaves, since we know that they are going to be deleted at a later stage during the planarization algorithm. Note that Pref(X) will be removed from T_i during the vertex addition stage of the reduction algorithm, so that any empty leaves in the sequence will be ignored. This guarantees the planarity of the resulting subgraph, since the reduction is then valid. We have the following result on the complexity of the modified compute numbers algorithm.

Lemma 4.6: The complexity of the modified compute numbers algorithm is $O(p^2)$.
Proof: The proof follows directly from the complexity of the compute numbers algorithm, and by observing that we introduce no further complexity measures. ❑

Once we have completed the modified compute numbers algorithm, we may proceed to assign a label to Root(T_i, S_i), and hence assign labels to the descendants of Root(T_i, S_i). The delete nodes algorithm is completely unchanged from the previous subsection, except that the discussion for

the delete nodes algorithm now refers to our new full and empty definitions.

The last major modification to the algorithm is the bubble pass. Note that the modified compute numbers algorithm needs to be able to determine whether, for a node X, there are any empty leaves representing edges of G_p in Frontier(X). That is, we need to classify nodes as full, empty or partial. Recall from the preceding discussion that this is an important requirement. If X only has empty leaves in its frontier, then the bubble pass will not even visit this node, since the bubble pass procedure only considers nodes from Pruned(T_i, S_i). Thus, we have no way of telling if Frontier(X) has empty nodes from G_p. Further, since we sometimes delete nodes from T_i in the vertex addition stage, it may happen that Frontier(X) has, for some reduction for vertex j, leaves from G_p, but in a later reduction those leaves were deleted and Frontier(X) no longer has any leaves from G_p. A problem arises when trying to determine if, for a node X, an empty child Y has leaves in Frontier(Y) from $E(G_p)$.

Previously the bubble pass consider only pertinent leaves and their ancestors. We now modify the bubble pass to consider, as well as pertinent leaves, all empty leaves representing edges of G_p in T_i. We refer to the modified algorithm as the *modified bubble pass algorithm*. We have the following interesting result on the complexity of the algorithm.

Lemma 4.7: The modified bubble pass algorithm has overall complexity $O(p^2)$.

Proof: Denote by $n_p(T_i)$ the total number of leaves in T_i belonging to G_p, and let Unary(T_i) be the number of unary nodes (i.e. nodes having only one pertinent child), except the leaves, traversed by the modified bubble pass algorithm. As from Section 2.5, the complexity of the modified bubble pass algorithm for T_i is $O(n_p(T_i) + \text{Unary}(T_i))$. But, note that $n_p(T_i)$ is $O(|(E(G_p)|)$, and Unary(T_i) is O(p). Now, since G_p is planar, $|(E(G_p)|$ is O(p). Hence the complexity of the modified bubble pass algorithm for T_i is O(p). Thus, over all reductions for vertex i ($2 \leq i \leq p-1$), we get that the modified bubble pass algorithm has complexity $O(p^2)$. ❑

We obtain the following algorithm which finds a maximal planar subgraph of a non-planar graph G, if we are given a 2-connected planar subgraph H of G.

Algorithm 4.4: Maximal_Planarize_Jaya_Thul_Swamy

{ Determine maximal planar subgraph of a non-planar 2-connected graph G }

```
Planar_Subgraph(G)          { use Algorithm 4.3 }
                            { now check if feasible to proceed }
Test Gp for cut-vertices
if Gp has a cut-vertex      { is it 2-connected }
    then
        Exit                { quit this algorithm }
    else
        Construct initial PQ-tree T2
        for i = 2 to p-1 do
            Modified_Bubble_Up (Ti)          { get parent pointers etc }
            Modified_Compute Numbers (Ti)    { calculate ne, np, nm }

            if min (np, nm) of Root(Ti, Si) ≠ 0    { delete any edges ? }
                choose min (np, nm) and label Root(Ti, Si) accordingly
                delete nodes                       { delete edges if so }
                delete new pertinent leaves not in Ti { delete bad leaves }

            Reduce (Ti)             { perform reduction }
                                    { and do vertex addition stage }
            Remove Full Children and descendants
            Add new leaves being edges directed out of vi
            Update Descendants Count
```

end

All that remains is to show that Algorithm 4.4 does produce a maximal planar graph.

Theorem 4.4: Given a 2-connected non-planar graph G, and assuming that Algorithm 4.3 produces a 2-connected planar subgraph G_p of G, Algorithm 4.4 produces a maximal planar subgraph G' which contains G_p as a subgraph.

Proof: Since the reduction process has succeeded, G' is planar, and because we always included all preferred leaves representing edges from G_p, we know that G_p is a subgraph of G'.

Assume that G' is not maximal, and that there exists an edge e = j, k, for j < k, such that e $\notin$ E(G'), and G' $\cup$ {e} is planar. Among all such edges e, let e_1 be an edge such that k is a minimum, say $e_1 = j_1, k_1$.

Now, suppose that we modify Algorithm 4.4 to find a maximal planar subgraph of G with G' as an underlying planar subgraph (i.e. we do not use Algorithm 4.3 to find G_p, rather we use G' instead). We denote the PQ-tree constructed before the bubble pass for vertex i begins during running of the modified Algorithm 4.4 by T'_i. We denote the PQ-tree constructed before the bubble pass for vertex i begins during the normal running Algorithm 4.4 by T_i.

Now, consider the PQ-tree T_{k_1} with respect to G_p, which was present before the reduction for vertex k_1. Note that e_1 must be represented by a new pertinent leaf of T_{k_1} with respect to G_p. Further, e_1 was not added by Algorithm 4.4 to G'. Thus we may also consider the PQ-tree T'_{k_1} produced by Algorithm 4.4 when we planarize G'.

Note that $T_{k_1} \equiv T'_{k_1}$, since they only differ with respect to e_1, which is a new pertinent leaf of both trees. Also, since $G_p \subseteq$ G', all preferred pertinent leaves of T_{k_1} are preferred pertinent leaves of T'_{k_1}, and some new pertinent leaves of T_{k_1} may be preferred pertinent leaves of T'_{k_1}.

Since G' $\cup$ {e} is planar, through a sequence of permutations and reflections, T'_{k_1} may be converted to an equivalent PQ-tree T''_{k_1}, such that its frontier contains a maximal sequence of leaves Pref2(X) such that

(i) if some of the leaves in Pref2(X) are empty then they appear at the endmost sides of Pref2(X) (i.e. the full children are consecutive);

(ii) Pref2(X) contains all preferred pertinent leaves of G';

(iii) Pref2(X) contains the new pertinent leaf representing the edge e_1, and possibly some other new pertinent leaves as well.

Now, since $T_{k_1} \equiv T'_{k_1}$, it follows that $T_{k_1} \equiv T''_{k_1}$ as well. Note that this sequence satisfies the requirements for the sequence Pref(X) in the modified compute numbers algorithm, where X is the pertinent root. But then, the leaves which belong to Pref(X) is a proper subset of Pref2(X),

since the leaf representing e_1 is not in the sequence Pref(X). This is a contradiction of the choice of Pref(X). ❑

The complexity of Algorithm 4.4 follows directly from earlier results. We have the following result.

Theorem 4.5: Algorithm 4.4 has complexity $O(p^2)$.

We next show that the check in Algorithm 4.4 that G_p is 2-connected is necessary. If the planar subgraph G_p is not 2-connected, then the original st-numbering is invalid. Furthermore, no valid st-numbering exists for G_p. Now, the PQ-tree reduction process relies on Lemma 2.11, which states that all the virtual edges of a bush form lie in one region of the graph constructed so far. This is now invalid because we allow more than one sink in the graph. Thus, if G' is the maximal planar subgraph produced by Algorithm 4.3, there may be another graph G" which contains G_p as a subgraph, and has more edges then G'. Hence, the restriction to 2-connected graphs is crucial.

Consider, as an example, the connected planar graph G, shown in Figure 4.5, below.

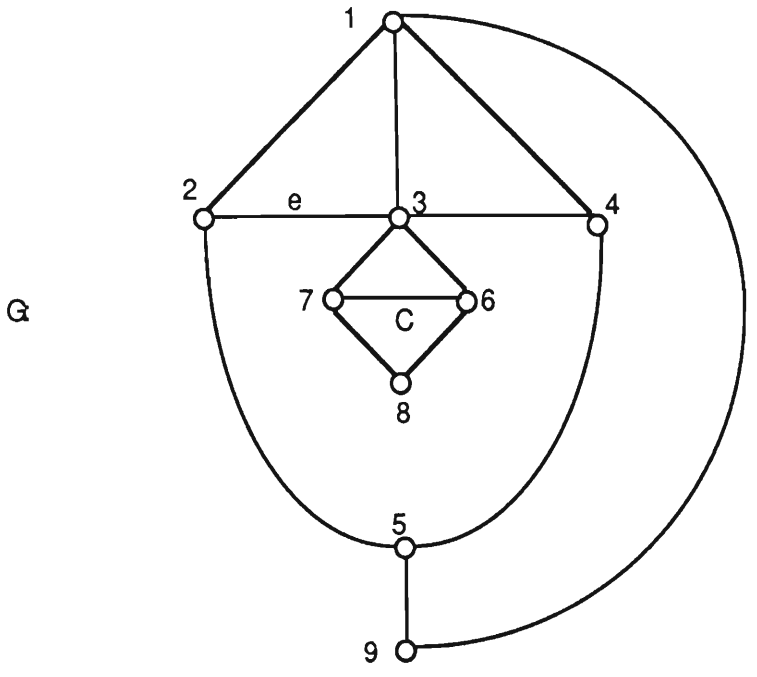


Figure 4.5 - An example graph with no st-numbering

During the reduction for vertex 5, Algorithm 4.4 will delete the edge 2, 5 or 4, 5, because there are edges from vertex 3 to vertices of higher st-number than 5. However, since both 8 and 9 are sinks, we may ignore these edges from vertex 3, and so we obtain a planar subgraph of G, with more edges than the planar subgraph G' produced by Algorithm 4.4. Thus, we have the strange situation that, given a planar graph G, Algorithm 4.4 could delete edges from G to find a maximal planar subgraph of G.

Section 4.2

Finding the Obstructions to Planarity

There are two known O(p) time algorithms for detecting a subgraph of a non-planar graph G which is a subdivision of the Kuratowski graphs K_5 or $K_{3,3}$. The first O(p) time algorithm is by Williamson [Wil84], and uses a Depth-First Search. We study here a more recent O(p) time algorithm by Karabeg [Kar88] which uses PQ-trees to detect a subdivision of the Kuratowski graphs.

Until now, we have only considered PQ-tree reduction algorithms for planar graphs. If we are to allow the basic PQ-tree reduction algorithm to detect the Kuratowski subgraphs, then it follows that we must consider templates to match the non-planar situations. Thus, we first consider an extension of the basic set of templates, the Templates P_0, P_1, ..., P_6, Q_0, Q_1, ..., Q_3, L_0, L_1 to cater for non-planar graphs. Karabeg has introduced four extra templates to match non-planar situations, which we name Templates N_1, N_2, N_3 and N_4. Templates N_1 and N_2 apply to Q-nodes, and Templates N_3 and N_4 apply to P-nodes. Note that in this section we are not concerned with replacement patterns of the templates.

Recall, for a node X in a PQ-tree T, Frontier(X) denotes the leaves of the maximal subtree of T rooted at X. Let X be a Q-node of T. Now, if in every PQ-tree T' equivalent to the maximal PQ-subtree rooted at X, the number of maximal sequences of consecutive pertinent leaves in Frontier(T') is at least 2, then Template N_1 is matched. One of the situations that satisfy Template N_1 is shown below in Figure 4.6.

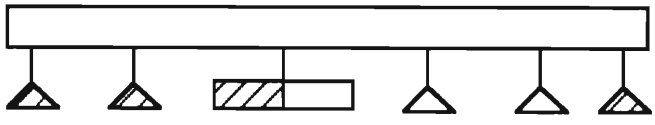


Figure 4.6 - Template N_1

Let X be a Q-node of T, where X is not the pertinent root. We say that X matches Template N_2 if X does not match Template N_1, and every PQ-tree T', equivalent to the maximal subtree rooted at X, in which the pertinent leaves of Frontier(T') appear consecutively, both the leftmost and rightmost elements of Frontier(T') are empty. One of the situations that satisfy Template N_2 is shown below in Figure 4.7.

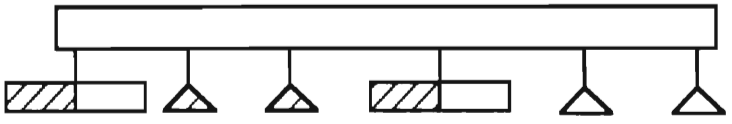


Figure 4.7 - Template $\mathcal{N}_2$

As we shall show later, Templates Q_0, Q_1, Q_2, Q_3, N_1 and N_2 are sufficient to describe every possible combination the children of a Q-node may be in. Note that Karabeg [Kar88] failed to correctly describe these cases. Template N_1 was described as "a Q-node with at least one interior child which is partial or empty, and which has at least one pertinent sibling on both sides". This description is incorrect, since a Q-node may now satisfy both Template Q_3 and Template N_1.

The templates which cater for non-planar cases when the node in question is a P-node are much easier. For Template N_3 to match we must have a P-node which is the pertinent root, with three or more partial children. Template N_3 is shown in Figure 4.8. For clarity we omit the rest of the partial children (if any), and the full and empty children (zero or more may be present) from the figure.

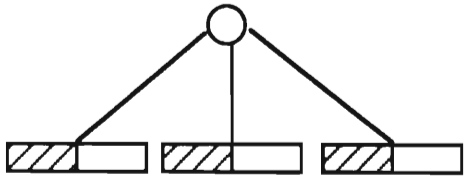


Figure 4.8 - Template $\mathcal{N}_3$

Template N_4 is similar to Template N_3, except that, in this case, the P-node in question may not be the pertinent root, and it must have at least

two partial children. Figure 4.9, below, illustrates this template, where, again, we have omitted the rest of the partial children, and the full and empty children from the figure.

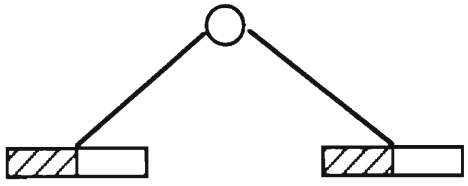


Figure 4.9 - Template $\mathcal{N}_3$

The next result shows that the Templates P_1, P_2, ..., P_6, Q_1, Q_2, Q_3, L_1, N_1, N_2, ..., N_4 cover all situations which may arise during template matching.

Theorem 4.6: The normal reduction process for vertex i fails at a node X of the PQ-tree T_i, if and only if one of Templates N_1, N_2, N_3 and N_4 is satisfied.

Proof: Firstly, we note that no template of the set Templates P_0, P_1, ..., P_6, Q_0, Q_1, ..., Q_3, L_0, L_1 is satisfied if one of Templates N_1, N_2, N_3 and N_4 is satisfied. Thus, if one of Templates N_1, N_2, N_3 and N_4 is satisfied, then the PQ-tree is irreducible.

Assume now that the reduction process fails, i.e. none of the Templates P_1, P_2, ..., P_6, Q_1, Q_2, Q_3, L_1 can be matched, and that none of Templates N_1, N_2, N_3 and N_4 can be matched.

Case 1: Suppose that X is a P-node.

Subcase 1.1: Further, suppose that X is the pertinent root. Then, since Template N_3 was not satisfied, X must have at most two partial children, with all the other children being full or empty. However, in this case, if we have two partial children, Template P_6 can be matched, or there is exactly one partial child and Template P_4 can be matched, or there are no partial children and Template P_1 or Template P_2 can be matched. In all the cases we obtain a contradiction to the assumption.

Subcase 1.2: Suppose that X is not the pertinent root. Then, since Template N_4 cannot be matched, X has at most one partial child. However, if X has exactly one partial child, Template P_5 can be matched,

and if X has no partial children, then Templates P_3 or P_1 can be matched. In all these cases we obtain a contradiction to the assumption.

Case 2: Suppose that X is a Q-node.

If Template N_1 does not apply, then X has only one sequence of successive full children in some PQ-tree T' equivalent to T_i. If this sequence includes an endmost child, and the endmost child is full, then either Template Q_1 or Template Q_2 can be matched. Thus, we may assume that both endmost children of X are either empty or partial, and that X has only one sequence of successive full children in some T' equivalent to T.

Subcase 2.1: Suppose that X is the pertinent root. Then Template Q_3 can be matched, contrary to assumption.

Subcase 2.1: Suppose that X is not the pertinent root. Then Template N_2 would apply, contrary to the assumption.

Thus, in all cases we obtain a contradiction, so one of the Templates N_1, N_2, N_3 and N_4 would be satisfied if the reduction process failed. ❑

As a reference to the Triconnectivity Algorithm of Section 3.3, note that, since only full nodes are used to detect separation pairs, we are able to extend our triconnectivity algorithm to non-planar graphs. For the Triconnectivity Algorithm, the replacement patterns for Templates N_1, N_2, N_3 and N_4 are described in [Kar89].

We define the subgraph G_i of a graph G to be the subgraph constructed by the PQ-tree algorithm before reduction for vertex i commences. Consider Figure 4.10, showing a planar embedding of a 2-connected subgraph G_{18} of a graph G, as well as some virtual edges. In the corresponding PQ-tree T, the 2-connected subgraph is represented by a Q-node. Note that not all of G_{18} is shown, and that not all virtual edges are shown.

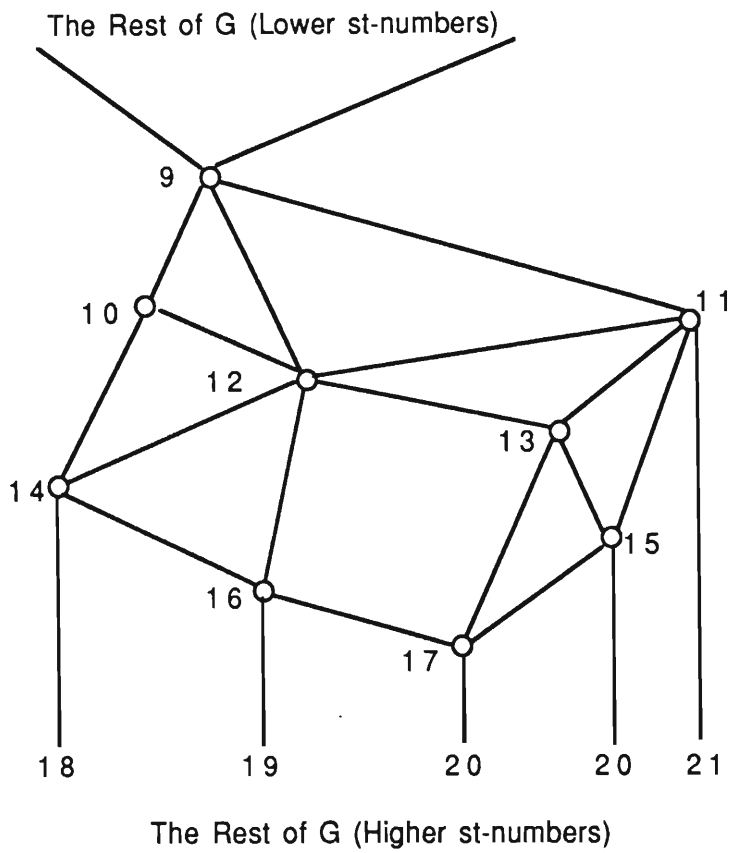


Figure 4.10 - A 2-connected subgraph of G_{18} with virtual edges

For a reduction for vertex i, we define a *triangle* to be a maximal 2-connected subgraph of G_i induced by a set of vertices attached to the rest of G_i by a single vertex. Given a triangle T of G_i, the vertex with minimum st-number in V(T) is called the *joint* of T (this is the same as the joint of a Q-node as defined in Section 3.3). The outer facial cycle of T is called the *boundary* of T. Note that the joint of T always belongs to the boundary of T. The first two vertices u and v encountered by starting at the joint of T, and proceeding along the boundary of T on the two paths of the boundary, such that not all of u and v's adjacent vertices belong to T are called the *endmost vertices of T*. Consider the path P along the boundary of T with the end vertices of P being the two endmost vertices, and such that the joint is not on P. All internal vertices of P are called *interior vertices*. Figure 4.11, below, illustrates these concepts for the triangle shown in Figure 4.10.

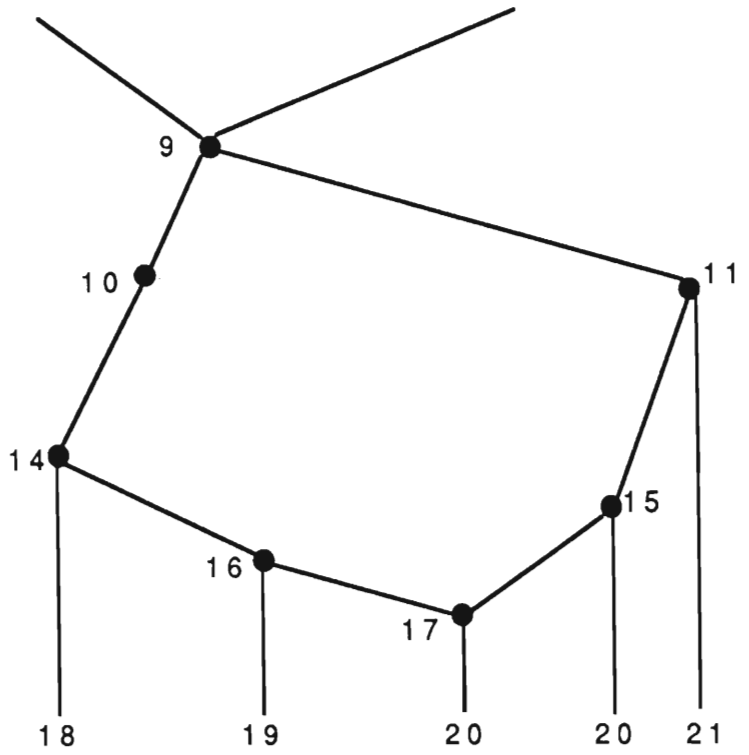


Figure 4.11 - The outer facial cycle of a triangle T, with virtual edges the joint is 9; 11 and 14 are the two endmost vertices; 16, 17 and 15 are the interior vertices.

Figure 4.12, below, illustrates the Q-node corresponding to the triangle shown in Figure 4.11.

Figure 4.12 - Q-node corresponding to the triangle in Figure 4.7

For every Q-node we shall maintain additional information specifying the boundary of the corresponding triangle, which we denote, for a Q-node X, by Boundary(X). An important point is to note that the children of X represent the vertices of Boundary(X) which have virtual edges. Note also, as in Section 3.3, that for every P-node we maintain the label of that node, and for every Q-node we maintain the joint of that node.

Now, from each template match of the new templates, Templates N_1, N_2, N_3 and N_4, we may extract the corresponding subgraphs which are subdivisions of the Kuratowski graphs. We have the following algorithm.

Algorithm 4.5: Detect_Kuratowski_Subgraphs
{ Given non-planar graph G, detect Kuratowski subgraphs }

```
If a node X fails the template matching pass (Algorithm 2.13)
    then
        if Template_N1(X)
            then
        { get vertex and edge set for subdivision of K3,3 }
                Vertex_Set_N1(X)
                Edge_Set_N1(X)
        else if Template_N2(X)
            then
        { get vertex and edge set for subdivision of K3,3 }
                Vertex_Set_N2(X)
                Edge_Set_N2(X)
        else if Template_N3(X)
            then
        { get vertex and edge set for subdivision of K3,3 }
                Vertex_Set_N3(X)
                Edge_Set_N3(X)
        else if Template_N4(X)
            then
        { get vertex and edge set for subdivision of K3,3 or K5}
                Vertex_Set_N4(X)
                Edge_Set_N4(X)
```

end

The template matching code for Templates N_1, N_2, N_3 and N_4 is straightforward, and we only describe the algorithms briefly. Note that, when Algorithm 4.5 is called for a node X, we have the lists Full_Children and Partial_Children, as described in Section 2.5. For Template N_1, we merely scan through the list Full_Children, and, by traversing the immediate sibling chains, we can easily detect if all full children are consecutive. For Template N_2, we note that, by Theorem 4.6, if X is a Q-node, then, since Template N_1 failed, Template N_2 is automatically satisfied. Template N_3 matches if X is the pertinent root (again, by Theorem 4.6, by now we know that X must be a P-node), and Template N_4 matches if X is not the pertinent root.

We now present the algorithms to determine the vertex and edge sets of a subdivision of either K_5 or $K_{3,3}$. Note that, in the algorithms for determining the vertex sets of a subdivision of K_5 or $K_{3,3}$, we only identify which vertices belong to the relevant Kuratowski graph. In the case of $K_{3,3}$, we also stipulate to which partition of the vertex set of $K_{3,3}$ the relevant vertex belongs. First, we consider Vertex_Set_N_1 and Edge_Set_N_1. We have the following algorithm Vertex_Set_N_1. Note that, for the diagrams for the rest of this section, paths are represented by edges, where we do not explicitly draw the edges.

Algorithm 4.6: Vertex_Set_N_1(X)

{ Template N_1 has matched - detect vertex set of $K_{3,3}$ }

{ see Figure 4.13 }

By traversing interior children of X only,
proceed from both endmost children of X and
select the first two vertices M and N
that are represented by full children of X
Select any other interior vertex Y of Boundary(X) which is represented by an empty or partial child of X, and that lies between M and N in Boundary(X)

Mark joint of X by a cross
Mark M and N by a nought
Mark Y by a cross

Mark the vertex Z which we are reducing with respect to by a cross

Find the paths from Y and Z to p
If the paths are disjoint
then
mark p by a nought
else
mark first common vertex W on the paths by a nought
Output vertices marked by crosses as one partite set
Output vertices marked by noughts as one partite set

end

There are a number of principles from Algorithm 4.6 which we use in discussing the rest of the algorithms. Note that the existence of two sequences of successive full children of X guarantees that we may always find three distinct vertices M, N and Y in Boundary(X). Further, observe

how we may traverse the sibling chains of the children of X to obtain the desired vertices M, N and Y of Boundary(X). The paths from Y and Z are easily determined. To determine a path from Y (to W) which does not contain Z, we merely choose, as first edge in the path, an edge directed from Y, represented by an empty node in the PQ-tree. Then, by choosing each subsequent vertex along the path as a vertex with higher st-number than the current vertex, the st-numbering guarantees that we obtain the desired result. From Figure 4.13 it is easy to see that Algorithm 4.6 does indeed find a subdivision of $K_{3,3}$.

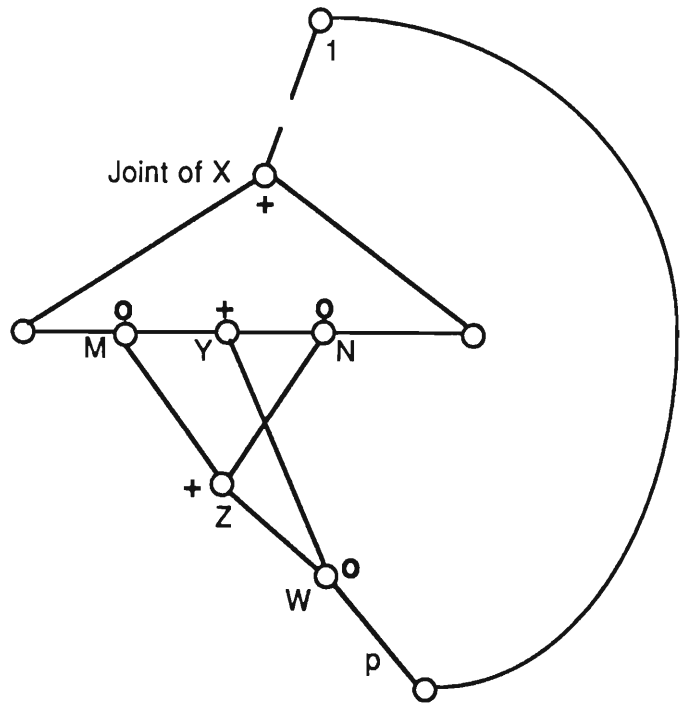


Figure 4.13 - Subdivision of $K_{3,3}$ when N_1 is matched

We may now consider the algorithm Edge_Set_N_1.

Algorithm 4.7: Edge_Set_N_1(X)

{ Find edge set to match vertex set found in Algorithm 4.6 }

Determine the two paths from the joint of X,
along Boundary(X) to M and N
Determine two paths along Boundary(X)
from M and N to Y
Determine two paths from M and N to Z
Determine paths from Y and Z to first common vertex W

```
        Determine path from W to p followed by edge p 1, and
            then the path from 1 to joint of X
        Output Edges of the subdivision
```

end

Note that, given Boundary(X), a traversal of Boundary(X) is easy. Again, in an analogous manner to the previous discussion, to determine a path from M and N to Z, we merely choose, as first edge along the paths, a full node in the PQ-tree from that respective vertex, and use the st-numbering.

The algorithm Vertex_Set_N_2 is very similar to Algorithm 4.6. We present it below.

Algorithm 4.8: Vertex_Set_N_2(X)

{ Template N_2 has matched - detect vertex set of $K_{3,3}$ }

```
        { see Figure 4.14 }

    Mark joint of Q-node X by a cross
    Mark a vertex W represented by a pertinent interior node of X by a cross
    Mark the two endmost vertices Y and Z of Boundary(X) by noughts

    Mark the vertex K which we are reducing with respect to by a nought

    Find the paths from Y and Z which
        each pass through an empty descendant to p,
        and a path from K to p
    If the paths are disjoint
        then
            mark p by a cross
        else
            mark first common vertex which any
                two or three of the paths have by a cross
    Output vertices marked by crosses as one partite set
    Output vertices marked by noughts as one partite set
```

end

Notice that we require that the vertex W must be an interior vertex. This is because we select the endmost vertices of Boundary(X) to be in the other partite set of $K_{3,3}$. Again, once the vertex set for Template N_2 has been determined by Algorithm 4.8, the algorithm to determine the corresponding edge set is easy.

Algorithm 4.9: Edge_Set_N_2(X)

{ Find edge set to match vertex set found in Algorithm 4.8 }

Determine two paths along Boundary(X)
from joint of X to Y and Z
Determine two paths along Boundary(X)
from Y and Z to W
Determine path from W to full descendant K
Determine paths from K, Y and W to first common descendant,
as described in Algorithm 4.8
Determine path from the joint of X to pertinent root,
and hence via a full node, to K
Output Edges of the subdivision

end

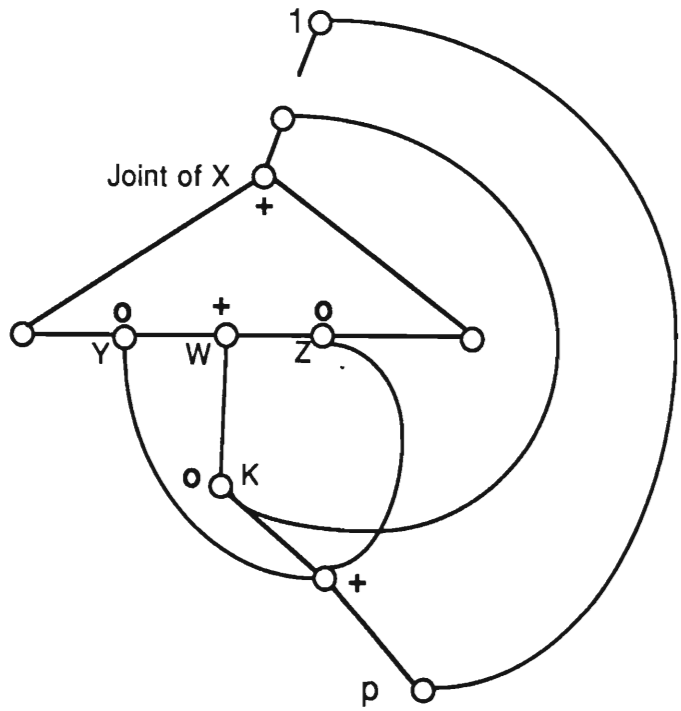


Figure 4.14 - Subdivision of $K_{3,3}$ when N_2 is matched

The cases when X is a P-node are a little more difficult. Algorithm 4.10, below, illustrates the case when X matches Template N_3.

Algorithm 4.10: Vertex_Set_N_3(X)

{ Template N_3 has matched - detect vertex set of $K_{3,3}$ }

```
        { see Figure 4.15 }

    Choose three partial children of X
    From each these partial children
        choose a vertex represented by a full node
    Denote the three vertices thus selected by Y, Z and W

    Mark Y, Z and W by a nought
    Mark the vertex having st-number the label of X by a cross
    Mark the vertex U which we are reducing with respect to by a cross

    Find the paths from Y and Z and W not passing through U to p
    If the paths are disjoint
        then
            mark p by a cross
        else
            mark first common vertex K which any
                two or three of the paths have by a cross
    Output vertices marked by crosses as one partite set
    Output vertices marked by noughts as one partite set
```

end

We note the same comment about the path determination as before. By choosing an empty child as first edge on the paths from Y and Z to the first common vertex K, the desired paths are constructed. Again, the determination of the edge set corresponding to the vertex set determined in Algorithm 4.10 is easy to see. We have the following algorithm.

Algorithm 4.11: Edge_Set_N_3(X)

```
        { Find edge set to match vertex set found in Algorithm 4.10 }

    Determine the three paths from the vertex
        represented by the label of X to Y, Z and W
    Determine three paths from Y, Z and W to U
    Determine three paths from Y, Z and W to K not passing through U
    Output Edges of the subdivision
```

end

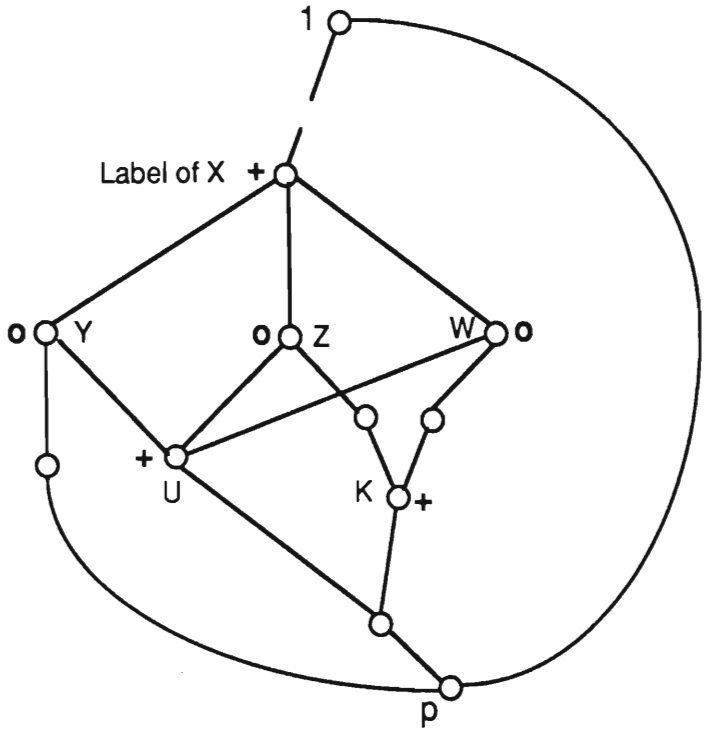


Figure 4.15 - Subdivision of $K_{3,3}$ when N_3 is matched

The final case we consider is if a P-node X matches Template N_4. This case is more complicated, and we find that we may have a subdivision of K_5 or of $K_{3,3}$. Algorithm 4.12, below, gives the algorithm. Note that, in Algorithm 4.12, we detect a subdivision of K_5 as a special case. This case occurs when the label of X is the same as the label of its parent. In this case we can deduce that, at some stage in the algorithm, X was a Q-node with only 2 children. To keep the PQ-tree proper, we modified the PQ-tree and changed X to a P-node.

Algorithm 4.12: Vertex_Set_N_4(X)

{ Template N_4 has matched - detect vertex set of $K_{3,3}$ or K_5 }

{ see Figures 4.16, 4.17 and 4.18 }

Choose two partial children of X
Denote the vertices Y and Z corresponding to the
empty endmost child of the chosen partial children of X

if X has label different to label of Parent(X)
then { we have a $K_{3,3}$ case }

```
                { see the discussion to follow }
                { and see Figure 4.16 }

            Mark Y and Z by a nought
            Mark the vertex corresponding to the label of X by a cross
            Mark root of pertinent subtree by a nought
            Mark the vertex W which we are reducing with respect to by a
cross

            Find the paths from Y and Z not through W to p
            If the paths are disjoint
                then
                    mark p by a cross
                else
                    mark first common vertex K which
                        the paths have in common by a cross
            Output vertices marked by crosses as one partite set
            Output vertices marked by noughts as one partite set

        else                    { we have either K3,3 or K5 }
                { X has same label as Parent(X) }
            Let W be the vertex which we are
                reducing with respect to
            If X does not have the same label as the pertinent root
                then
                        { see Figure 4.16 }
                    Mark pertinent root with a nought
                    Mark label of X with a cross
                    Mark Y and Z with noughts
                    Mark W with a cross
                    Mark first common vertex K on the
                        paths from Y to p and from Z to p with a cross
                    Output vertices marked by crosses as one partite set
                    Output vertices marked by noughts as one partite set

                else    { X has same label as pertinent root }
                    Let P1 be the Y - p path which contains an empty child
                    Let P2 be the Z - p path which contains an empty child
                    Let P3 be the path from W to p
                    Let K be the first vertex any two of these paths
                        have in common
                    If all three paths have K in common
                        then                { we have a K5 case }
                                { See Figure 4.17 }
                            Mark the vertex corresponding to the label of X,
                            and the vertices Y and Z,
                            and the vertex we are reducing with respect to,
                            and vertex K with noughts
                            Output vertices marked by noughts as
                                vertices of K5

                        else                { we have a K3,3 case }
                                { See Figure 4.18 }
```

```
Mark the joint of X by a cross
Mark vertex K with a cross
Mark the end vertices (which belong to {W, Y, Z})
    of the two paths which intersect in K
        with noughts
Mark the remaining vertex in {W, Y, Z}
        by a cross
Mark the first common vertex N on the path
    from the vertex in {W, Y, Z}, which is
        marked with a cross, to p,
    and on the path from K to p
        with a nought
Output vertices marked by crosses
    as one partite set
Output vertices marked by noughts
    as one partite set
```

end

Note that a node sometimes may have the same label or joint as its parent. In Algorithm 4.12, we must check, although X is not the pertinent root, the label of X is not the same as the label or joint of the parent. This check ensures that we obtain a valid subdivision of $K_{3,3}$. If the label of X is the same as the label of the parent, then we note that X was a Q-node with two children. The PQ-tree algorithm required that the PQ-tree was kept proper, and so we modified the tree so that X became a P-node with 2 children. Now, in this case we may obtain a subdivision of K_5, and not $K_{3,3}$ as was previously found. Thus, if X was a Q-node, at some stage, then X represents a 2-connected subgraph. So, in terms of X as a Q-node, Y and Z have a path connecting them along the path of Boundary(X) which does not proceed via the joint of X, as in Figure 4.17.

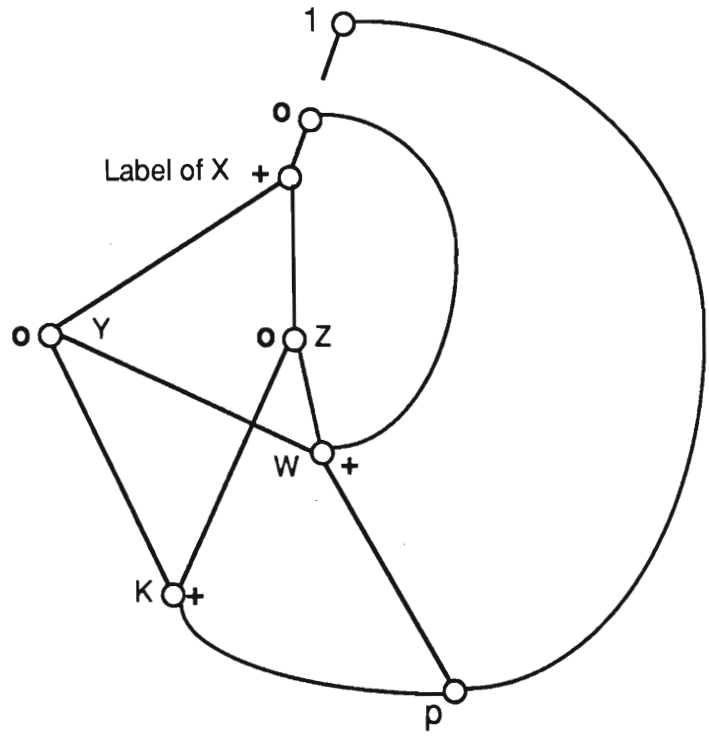


Figure 4.16 - Subdivision of $K_{3,3}$ when N_4 is matched

Algorithm 4.13: Edge_Set_N_4(X)

{ Find edge set to match vertex set found in Algorithm 4.6 }

```
if X has label different to joint or label of Parent(X)
    then        { K3,3 case }
        Determine paths from the label of X to Y and Z
        Determine path from root of pertinent subtree to the label of X
        Determine paths from Y and Z to W    { see comment, below }
        Determine paths from Y and Z to K
        Determine path from pertinent root to W not via label of X
        Determine path from K to p and hence
            via edge e = 1 p to pertinent root
    else
        If X does not have the same label as the pertinent root
            then        { K3,3 case }
                Find paths from the label of X to Y and Z
                Find path from root of pertinent subtree to the label of X
                Find paths from Y and Z to W      { see comment, below }
                Find paths from Y and Z to K
                Find path from pertinent root to W not via label of X
                Find path from K to p and hence
                    via edge e = 1 p to pertinent root

            else
```

```
If the three paths all have K in common
    then        { K5 case }
        Determine paths from the label of X to Y and Z
        Determine path from pertinent root
            to W not via Y or Z
        Determine path from Y to Z
        Determine paths from Y and Z to W
        Determine paths from Y, Z and W to K
        Determine path from K to p and hence via
            edge 1 p to pertinent root

    else        { K3,3 case }
        Determine paths from the label of X to Y and Z
        Determine path from pertinent root
            to W not via Y or Z
        Determine path from Y to Z
        Determine paths from Y and Z to W
        Determine paths from the two vertices of
            Y, Z and W which have paths to K
        Determine paths from K to N and the
            other vertex of Y, Z and W to N

Output Edges of the relevant subdivision
```

end

Notice that we may obtain paths from Y and Z to W by proceeding along the siblings of the particular partial child's children until we encounter a full child. We then select this edge for our path, and proceed as usual with our path determination algorithm.

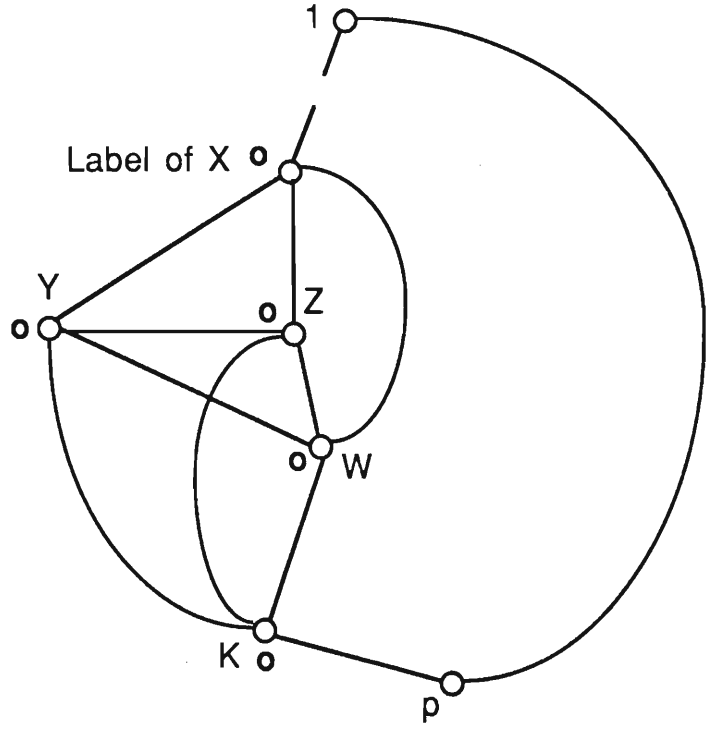


Figure 4.17 - Special case of Template $\mathcal{N}_4$ subdivision of $\mathcal{K}_5$ is found

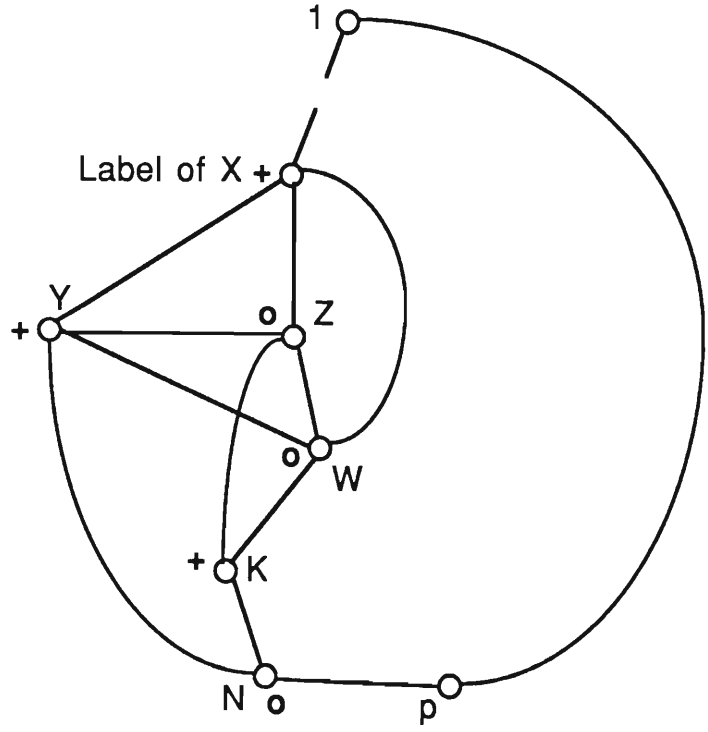


Figure 4.18 - Special Case of Template $\mathcal{N}_4$ subdivision of $\mathcal{K}_{3,3}$ is found

We have the following theorem describing the complexity of the overall algorithm to determine an obstruction to planarity, Algorithm 4.5.

Theorem 4.7: Algorithm 4.5 correctly determines either a subdivision of K_5 or $K_{3,3}$, and has O(p) time complexity.

Proof: From the construction of the subgraphs found, it is easy to see that Algorithm 4.5 does indeed find a subdivision of K_5 or $K_{3,3}$.

For the linear complexity, we need a bit more detail. Firstly, as discussed earlier, we may match Templates N_1, N_2, N_3 or N_4 in linear time through using the fields Partial_Children and Full_Children.

The maintenance of the label and joint values for each node in the PQ-tree have already been discussed, in detail, in Section 3.3.

We now show that, for a Q-node X, the correct maintenance of Boundary(X), during the normal reduction algorithm is possible in O(p) time.

For every Q-node X, we store Boundary(X) as a doubly linked list. In addition, we have a few special pointers into the doubly linked list. We have a pointer *Joint_Ptr* to the entry in Boundary(X) to the joint of X. Also, there are two pointers, called *Endmost1* and *Endmost2,* which point to the two endmost vertices in Boundary(X). Lastly, there are two pointers *Full1* and *Full2* which point to the two full children in the list Full_Children, with only one full immediate sibling. If there is only one full child then Full1 = Full2. If there are more than two full children with only one immediate full sibling, then Full1 and Full2 are undefined. Note that this case will never occur if Templates N_1, N_2, N_3 or N_4 are not matched.

Now, note that Templates P_1, Q_1 and P_2 do not require any maintenance of the above pointers. Thus, we may restrict our discussion to Templates P_3, P_4, P_5, P_6, Q_2 and Q_3. Suppose, for example, that we are matching Template Q_2 to a Q-node X. If X has no partial child, then Boundary(X) does not need updating, since the replacement pattern merely labels X as partial. On the other hand, if there is a partial child, then some work needs to be done.

Consider Figure 4.19, below, illustrating a subgraph corresponding to a possible match for Template Q_2 when we are reducing with respect to vertex 14. In Figure 4.19, for a Q-node Y, we refer to Y by its joint.

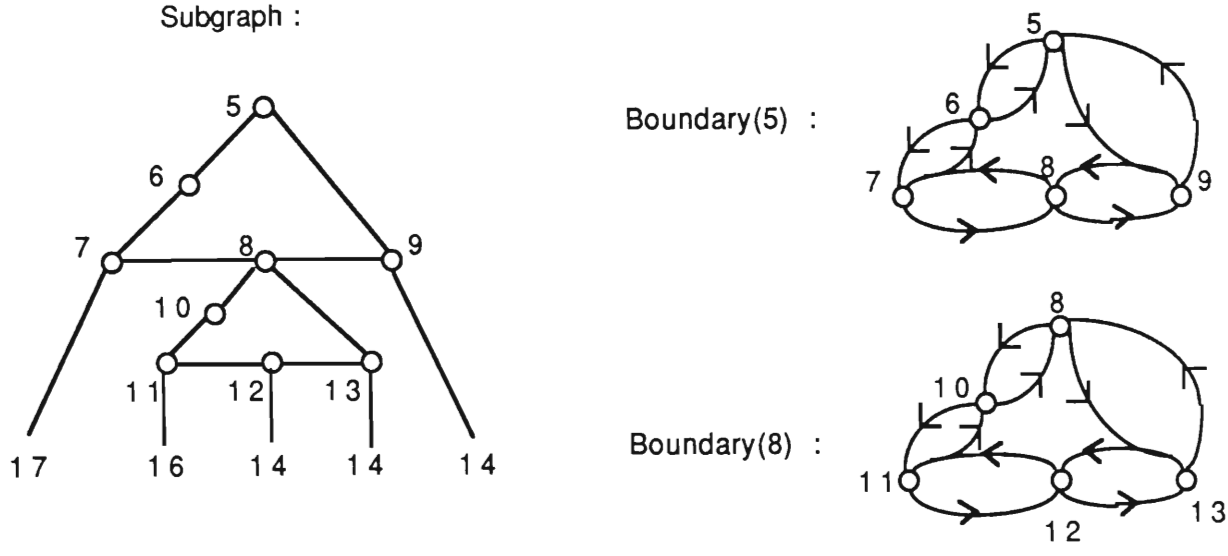


Figure 4.19 - A subgraph and corresponding data structures

After Template Q_2 has been matched, and the replacement pattern has been made, we need to merge the two doubly linked lists representing the two boundaries. This is because the replacement pattern specified that the two partial Q-nodes must merge into one partial Q-node. Thus, the two 2-connected subgraphs are merged. We must therefore update our description of the 2-connected subgraph.

Since the partial child must appear to one side of the consecutive sequence of full children, the element pointed to by Full1 or Full2 of Boundary(X) is adjacent to the element pointed to by Joint_Ptr of Boundary(Partial_Child(X)), say Full1 points to the adjacent element. Further, let Temp denote the element in Boundary(X) pointing to the vertex which corresponds to Joint_Ptr of Boundary(Partial_Child(X)). Thus, in the above example Temp points to vertex 8 in Boundary(5), and Full1 points to vertex 9.

Now, we may form the new doubly linked list as follows. We break the link of Boundary(X) at Temp, creating a new link between Full1 and the adjacent element of Joint_Ptr of Partial_Child(X) which points to a full endmost child of the partial child (in our above example this is to vertex 13). Then, we create a new link between the other adjacent element of Temp (i.e. not Full1) and the other adjacent element of Joint_Ptr of

Partial_Child(X). Temp is then deleted, since we have a duplicate reference to the joint of the partial child. Figure 4.20, below, shows the final result.

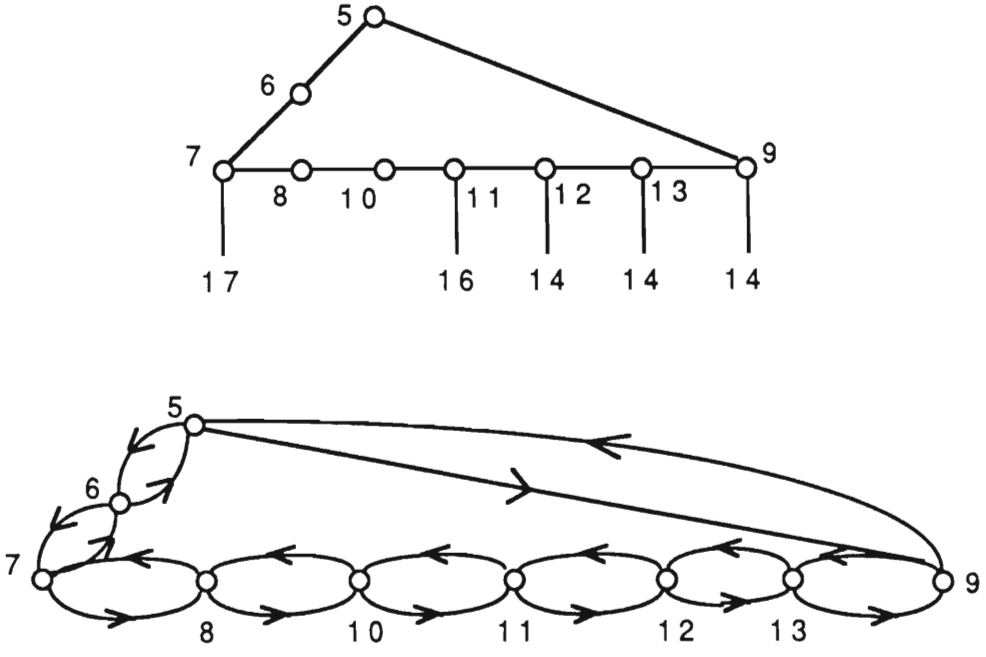


Figure 4.20 - A subgraph and corresponding data structure after replacement phase

It is easy to see that the four pointer manipulations correctly update Boundary(X), and that the complexity of the algorithm is still O(p) because Template Q_2 is matched O(p) times. Template Q_3 follows almost exactly the same process as described for Template Q_2. The only difference between the treatment of the two templates is that Template Q_3 may possibly contain two partial children, and so we would have to repeat the above procedure twice. Thus, since only four extra pointer manipulations are required, the complexity of the algorithm is still O(p).

For a P-node X, we require additional information giving the list of vertices of degree 2 between X and its children. Let the Q-node partial child be W. Then, when we match Template P_5, say, and we place the P-nodes with full and empty children at either side of the boundary of the W, we must also update the section of Boundary(W) between the joint and the relevant endmost vertex of Boundary(W), by adding the vertices of

degree two stored in the list of X. Since each vertex is only placed in one of these lists, and since the P-node is added to a Q-node at most once, the complexity of these extra steps is proportional to the number of vertices of degree 2 in the graph. Hence the complexity of the algorithm remains O(p). For Template P_3 and P_4, the process is exactly the same and requires no further discussion. For Template P_6, we note that if there are full children, we need to insert a new P-node between the two sequences of full children in the new partial node. To merge the boundaries of the two partial children we first discard the side of the triangles which has a full endmost child. Then, we merge the two lists, inserting between them the list of vertices of degree 2 between X and its children. It is not hard to see that, through the use of the Full1, Full2, Endmost1 and Endmost2 fields, one may easily perform the correct actions, which consist of a few pointer manipulations. Thus, since there are O(p) Template P_6 matches, the overall complexity of our algorithm remains O(p).

We have already discussed the algorithms to determine the paths, and it is not hard to see that the edge set determination algorithm has complexity O(p). Similarly, we get that the vertex set determination has complexity O(p).

Thus, we obtain the correct result. ❑

Section 4.3

A New Algorithm for finding an Upper Bound of the Genus of a Graph

Recall from Chapter 1 that the genus of a graph equals the sum of the genera of its blocks. Thus, it suffices to develop algorithms for finding the genus of a 2-connected graph. Determining the genus of an arbitrary 2-connected graph efficiently is an open problem (see Garey and Johnson [GJ79]). Thus, we consider in this section an algorithm for finding an upper bound on the genus of a 2-connected graph.

Suppose G is an arbitrary non-planar 2-connected graph. Using the PQ-tree data structure we obtain, at each stage of the reduction algorithm, a description of the partial embedding of G. Figure 4.21, below, illustrates an st-numbered non-planar graph G. That G is non-planar is easy to see since a subgraph of G isomorphic to $K_{3,3}$ is induced by the vertex set {1, 3, 4, 5, 6, 7}, with {1, 4, 6} and {3, 5, 7} being the two partite sets.

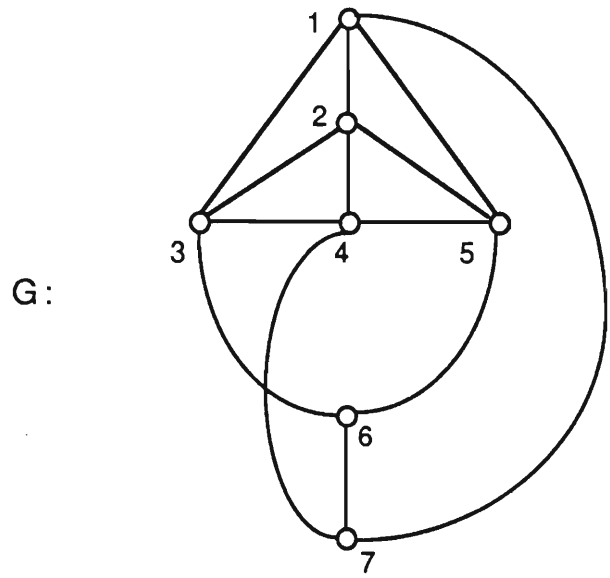


Figure 4.21 - An st-numbered non-planar graph G

Now, during the reduction algorithm, the reduction for vertex 6 fails, because both the pertinent leaves are children of a Q-node X, and they have an empty child (the leaf representing the edge from vertex 4 to vertex 7) between them, in the order as they appear as children of the Q-node. That is, X matches Template N_1. However, if we insert a handle with ends in regions R_1 and R_2, as shown in Figure 4.22(a), then we obtain an embedding of G on the torus, as shown in Figure 4.22(b), below.

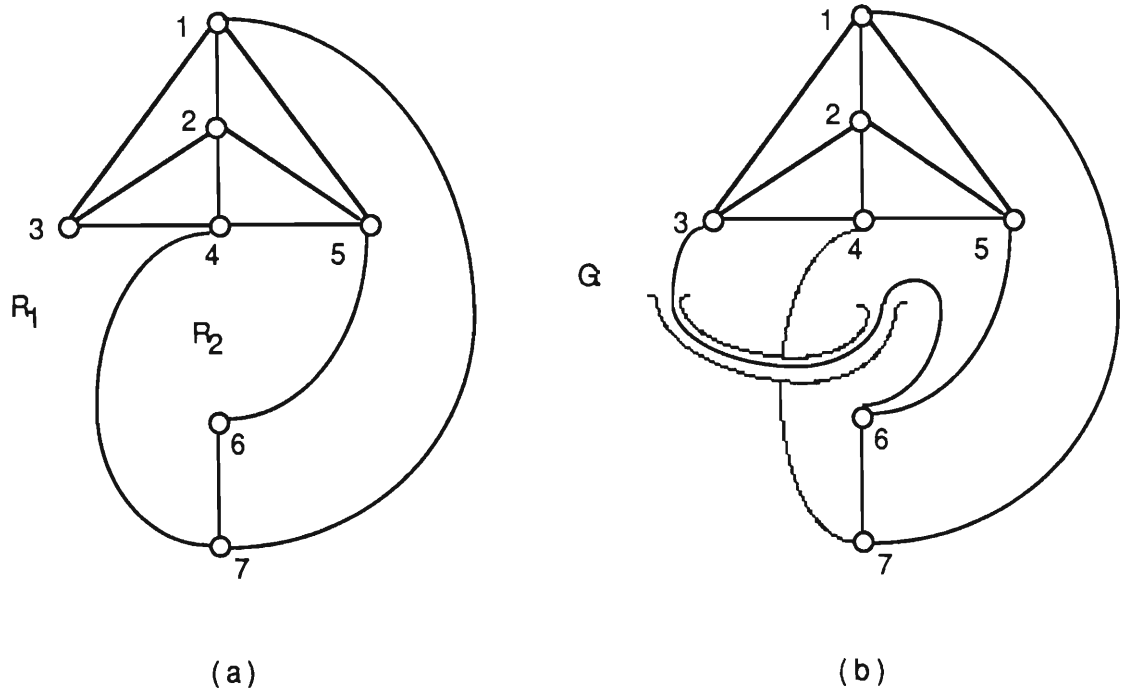


Figure 4.22 - An embedding of a graph G on the torus after inserting a handle on the sphere

The ideas of embedding the graph G of Figure 4.21 on the torus, as shown in Figure 4.22(b), hold the key to our algorithm. Observe that the embedding of G shown in Figure 4.22(b) is indeed a 2-cell embedding. Later we will show that our algorithm does not always produce a 2-cell embedding.

We now generalise the above idea to any non-planar graph. Observe that all we did in the above example was, given a template matching failure, to insert a handle from the one consecutive sequence of pertinent edges to a final region on whose boundary vertex 6 lies. This technique of

inserting a handle during the reduction process is the main idea behind our algorithm.

Suppose we are reducing with respect to some vertex with st-number i, and suppose that, during the reduction, a node X in our PQ-tree T matches one of Templates N_1, N_2, N_3 or N_4. Our goal is to delete maximal subsequences of full leaves from Frontier(X) to allow the reduction process to proceed. At the end of the reduction we only have one maximal subsequence of full leaves in Frontier(T). Given a maximal subsequence of full leaves to delete, we identify the edges represented by the full leaves in a *pseudo-vertex*. The single remaining subsequence of full leaves in Frontier(T), after the reduction is complete, is identified in a vertex called the *base*. We place a handle from each pseudo-vertex to the base. Finally we undo the edges identified at each pseudo-vertex and place the corresponding identified edges along that handle at the pseudo-vertex so that they are now identified at the base vertex.

Let T' be the PQ-tree equivalent to the maximal subtree rooted at a Q-node X such that in Frontier(X) there are as few maximal subsequences of full leaves as possible. Suppose that there are k maximal subsequences of full leaves in Frontier(T'). We define $Pert_i(X)$ to be the sequence of pertinent children of X whose full leaf descendants are precisely the i-th maximal sequence of full leaves in Frontier(T') as we proceed from left to right in Frontier(T').

We define an operation Reduce($Pert_i(X)$) to simplify $Pert_i(X)$ in the following manner. If there are no partial children in $Pert_i(X)$, then do nothing. Otherwise, for each partial child Y, merge the children of Y into the sibling lists of X so that the full or partial immediate sibling of Y in $Pert_i(X)$ is now an immediate sibling of the full endmost child of Y. Thus, the frontier of $Pert_i(X)$ still contains the same number of full leaves. Figure 4.23, below, illustrates the Reduce($Pert_i(X)$) operation.

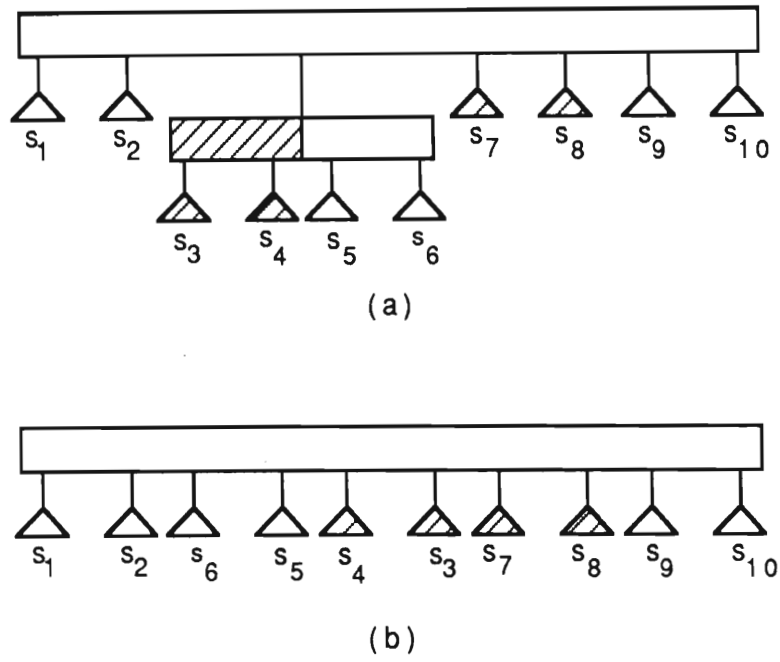


Figure 4.23 - The Reduce(Pert$_i$(X)) operation;
(a) before Reduce(Pert$_i$(X)); (b) after Reduce(Pert$_i$(X))

Using the Templates N_1, N_2, N_3 and N_4 introduced in the previous section, we may obtain the following algorithm which deals with a non-planar situation. For convenience we repeat, in comments in the algorithm, the requirements for every template to match.

Algorithm 4.14: Place_Handles_Non_Planar_Graph(X)

{ The template matching in PQ-tree T has failed
at a node X, we place handles to
allow the reduction to proceed }

```
If Template_N1(X)
    then    { Q-node, two or more isolated sequences of full children }
        Place_Handle_N1(X)
else If Template_N2(X)
    then    { Q-node, not pertinent root, no full child endmost }
        Place_Handle_N2(X)
else If Template_N3(X)
    then    { P-node, pertinent root, three or more partial children }
        Place_Handle_N3(X)
else If Template_N4(X)
    then    { P-node, not pertinent root, two or more partial children }
        Place_Handle_N4(X)
```

end

We denote by $\gamma_g(G)$, the number of handles which we place to embed G. Before we begin the reduction for vertex 2, we assume that we are to embed G on the sphere, i.e. $\gamma_g(G) = 0$.

We consider each of the cases in turn. Note that the algorithms may place a number of handles. Suppose we are reducing for vertex i. At first we shall only place the pseudo-vertices. The base can only be determined at the end of the reduction for vertex i. Suppose we have determined that the full leaves from a sequence $Pert_j(X)$ of pertinent children of X must be deleted. When we place a pseudo-vertex for some of the sequences $Pert_j(X)$, we shall mark the pertinent leaves which are in $Pert_j(X)$ as *void*. This marking process removes all references to the sequence of pertinent leaves from the Q-node X. This will then allow the reduction algorithm to continue the template matching process.

For the rest of this section, in the diagrams used to illustrate the different cases we encounter, we will use black circles to denote the ends of the handles which we insert, and hence the pseudo-vertex which we place.

Template N_1 is matched for a Q-node X, when we have at least two maximal subsequences of pertinent children in any Frontier(T'), where T' is equivalent to the maximal subtree of T rooted at X. The first operation we perform is Reduce($Pert_i(X)$) for every $Pert_i(X)$. Thus, our maximal subsequences of pertinent children are now only full children. We have two subcases, depending on whether if X is the pertinent root or not.

Case 1: Suppose that X is the pertinent root. Then, we select some subsequence $Pert_i(X)$. At the end of the reduction we will identify the edges represented by the pertinent leaves in the frontier of $Pert_i(X)$ in the base. For all other subsequences $Pert_j(X)$, we identify the pertinent leaves to a pseudo-vertex, and mark these leaves void.

Case 2: Suppose that X is not the pertinent root. In this case, if a full child is endmost, we select a sequence $Pert_i(X)$ containing that child. For all other sequences $Pert_j(X)$, we identify the pertinent leaves to a pseudo-

vertex, and mark these full leaves void. We do not identify the sequence $Pert_i(X)$ to a pseudo-vertex, because the reduction is now able to proceed without the placement of a further handle. In this way we minimise the number of handles placed.
Algorithm 4.15, below, gives the full algorithm.

Algorithm 4.15: Place_Handle_N_1(X)

{ Place Handle(s) to enable reduction to continue after reduction failed and X matched Template N_1}

```
    { Q-node, two or more isolated full sequences of children }
    { see Figure 4.24(a) }

Let the maximal sequences of pertinent children be
    Pert1(X), Pert2(X), ..., Pertk(X); where k ≥ 2
For all sequences Perti(X) do
    Reduce(Perti(X))
If X is not the pertinent root
    then
        If any full child Y of X is endmost
            then
                let Perti(X) be the sequence to which that
                    full child belongs
            else
                Perti(X) = ø
    else            { X is the pertinent root }
        Select any sequence Perti(X)

        { Perti(X) now contains a sequence of full children which we
            do not want to place a handle from }

for all sequences Pertj(X) ≠ Perti(X) do
    Identify Pertj(X) in a pseudo-vertex
    γg(G) = γg(G) + 1
    Mark Pertj(X) void          { i.e. delete Pertj(X) }

        { see Figure 4.24(b) }
{ we now have that X may be matched to one of
    Templates P0, ..., P6, Q0, ..., Q3 }
If X is not the pertinent root
    then
        Continue reduction and apply Template Q2
            or Template Q0 to X
        depending if Perti(X) ≠ ø or not
    else
        Continue reduction and apply Template Q2
            or Template Q3 to X,
```

depending if $Pert_i(X)$ is endmost or not

end

In Figure 4.24, we show a possible situation for some 2-connected non-planar graph G when we are reducing for vertex 17.

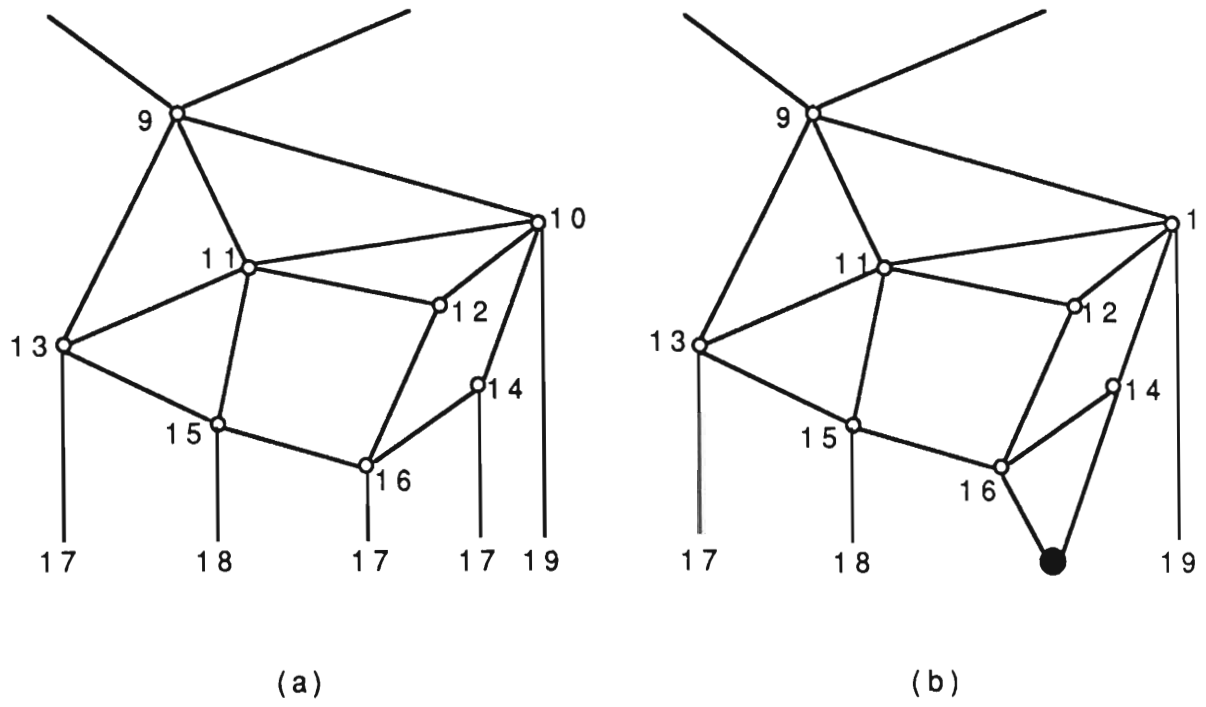


Figure 4.24 - Insertion of Handles when Template $\mathcal{N}_1$ is matched
(a) - Before reduction for vertex 17; (b) after placement of pseudo-vertex

We may now consider Algorithm Place_Handle_N_2(X), which caters for the case when the reduction process fails, Template N_1 does not match, and Template N_2 is matched. Since we have all pertinent children appear in one maximal sequence $Pert_i(X)$, it is sufficient to place a single handle. Again, we perform Reduce($Pert_i(X)$). Then, we identify all the pertinent leaves to a single pseudo-vertex. Algorithm 4.16, below, gives the full algorithm.

Algorithm 4.16: Place_Handle_N_2(X)

{ Place Handle(s) to enable reduction to continue after
reduction failed, Template N_1 does not match,
and X matched Template N_2}

```
        { Q-node, not pertinent root, no full child endmost }
        { see Figure 4.25 }

    Let the maximal sequence of pertinent children be Pert_i(X)
    Reduce(Pert_i(X))
    Identify Pert_i(X) in a pseudo-vertex
    γ_g(G) = γ_g(G) + 1
    Mark Pert_i(X) as void

            { see Figure 4.24(b) }
    { we now have that X may be matched to Templates Q_0 }
    { since all pertinent children have been deleted }
    Apply Template Q_0 to X
end
```

Figure 4.25, below, gives the diagram illustrating Template N_2. Once again, the psuedo-vertex is represented by a black circle, and we are reducing for vertex 17.

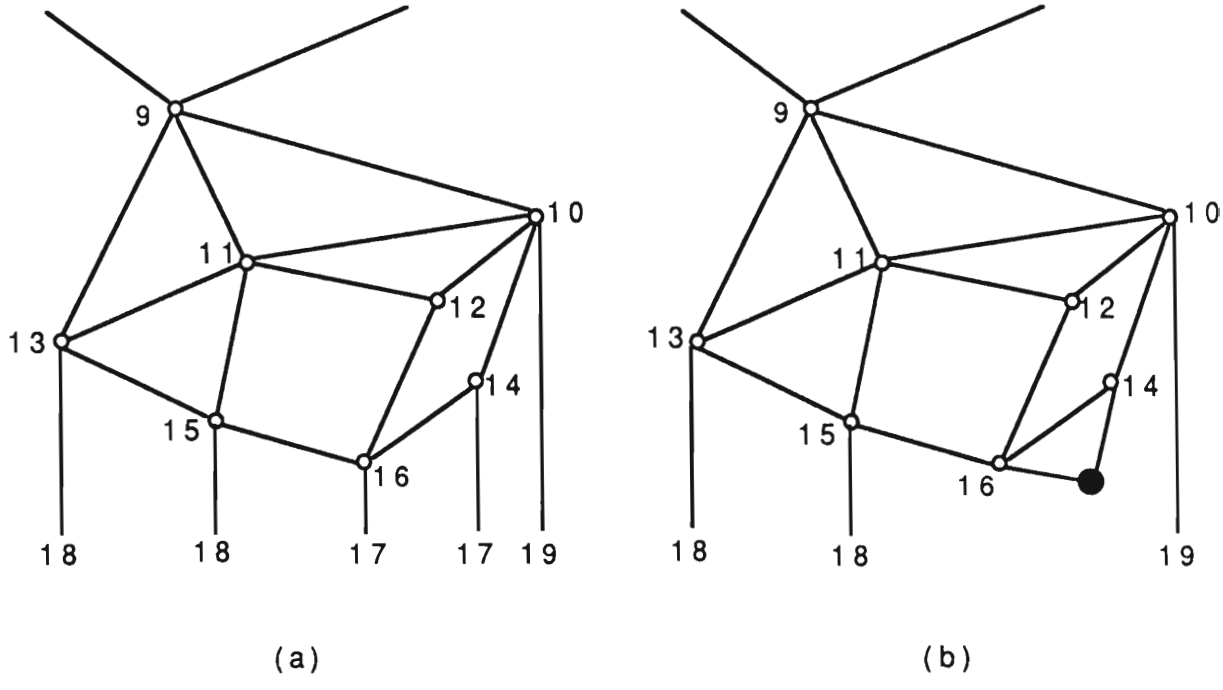


Figure 4.25 - Insertion of Handles when Template N_2 is matched
(a) - Before reduction for vertex 17; (b) after placement of pseudo-vertex

The rest of the algorithms deal with the Templates N_3 and N_4, which deal with P-nodes. Algorithm Place_Handle_N_3 uses a variation of the

replacement pattern for Template P_6. Suppose a P-node X satisfies Template N_3, so X must be the pertinent root. Further, suppose that there are k ($k > 2$) partial children of X. Let P_f and P_m be any two partial children of X. We place the full children of X between P_f and P_m. We then merge P_f and P_m to create a new Q-node Y with both endmost children empty, and all full children appearing in one consecutive subsequence $Pert_i(Y)$ of the children of Y. We then merge every other pair of partial children, thereby reducing the number of subsequences of full children to $\left\lceil \frac{k}{2} \right\rceil$. For every new partial child W and corresponding maximal subsequence $Pert_j(W) \neq Pert_i(Y)$, we identify the pertinent leaves of Frontier(W) in a pseudo-vertex, and mark all full children void. At the end of the reduction, $Pert_i(Y)$ is identified to our base.
Algorithm 4.17, below, gives Algorithm Place_Handle_N_3.

Algorithm 4.17: Place_Handle_N_3(X)

{ Place Handle(s) to enable reduction to complete after reduction failed and X matched Template N_3}

{ P-node, pertinent root, three or more partial children }
{ see Figure 4.26 }

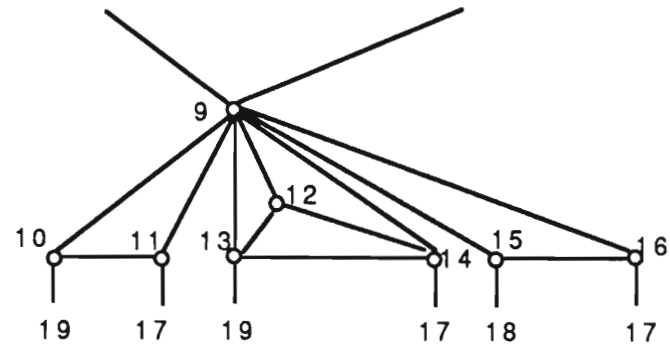
Let the partial children be $P_1, P_2, ..., P_k$; where $k \geq 2$
Denote the full and empty endmost children of any P_i by
Full(P_i) and Empty(P_i)

Partition the partial children into pairs.
Select any pair of partial children P_f and P_m
Remove any full children of X and place them
as children of a new P-node Y

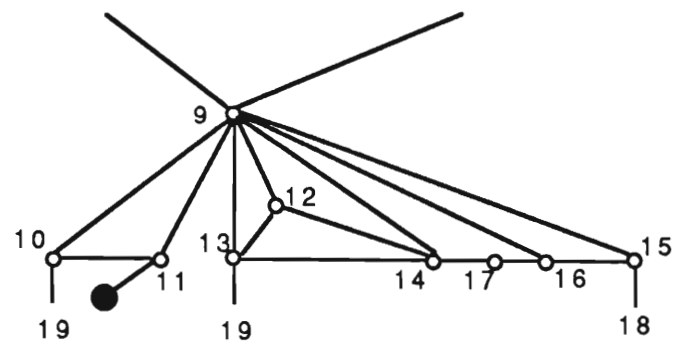
Join P_f and P_m together as follows
If Y has children
then
Add Y as an immediate sibling of
Full(P_f) and Full(P_m)
Make new endmost child of P_f be Empty(P_m)
else
Make Full(P_f) and Full(P_m) immediate siblings
Make new endmost child of P_f be Empty(P_m)
Denote full sequence of children of P_f by Main_Group

```
For every pair Pi and Pj (i, j ≠ ζ, m) do
    Make Full(Pi) and Full(Pj) immediate siblings
    Make new endmost child of Pi be Empty(Pj)
    Delete Pj
    Identify full children of Pi in a pseudo-vertex
    γg(G) = γg(G) + 1
    Mark full children of Pi as void
if k is odd
    then
        Identify Pk in a pseudo-vertex
        γg(G) = γg(G) + 1
        Mark full children of Pk as void
end
```

Figure 4.26, below, shows a subgraph of a non-planar graph G, when we are reducing for vertex 17.



(a)



(b)

Figure 4.26 - Illustration of Handles when Template $\mathcal{N}_3$ is matched
(a) - Before reduction for vertex 17; (b) after placement of pseudo-vertex

When Template N_4 is matched, the situation is almost the same as for Template N_3. Suppose a P-node X matches Template N_4, so X is not the pertinent root. Further, suppose that there are k ($k > 1$) partial children of X. Let P_ℓ and P_m be any two partial children of X. The full children of X must be placed between P_ℓ and P_m. We then merge P_ℓ and P_m to create a new Q-node Y with both endmost children empty, and all full children appearing in one consecutive subsequence $Pert_i(Y)$ of the children of Y. We then merge every other pair of partial children, thereby reducing the number of subsequences of full children to $\left\lceil \frac{k}{2} \right\rceil$. For every partial child W which does not have an endmost full child, and corresponding maximal subsequence $Pert_j(W)$, we identify the pertinent edges of Frontier(W) in a pseudo-vertex, and mark all full children void. Note that we avoid placing an extra handle if k is odd. If this situation occurs, then there is some sequence $Pert_i(X)$ which has a full endmost child. We do not identify the sequence $Pert_i(X)$ to a pseudo-vertex, because the reduction is now able to proceed without the placement of a further handle. Again, we attempt to minimise the number of handles placed. Algorithm 4.18, below, details the algorithm Place_Handle_N_4.

Algorithm 4.18: Place_Handle_N_4(X)

{ Place Handle(s) to enable reduction to continue after
reduction failed and X matched Template N_4}

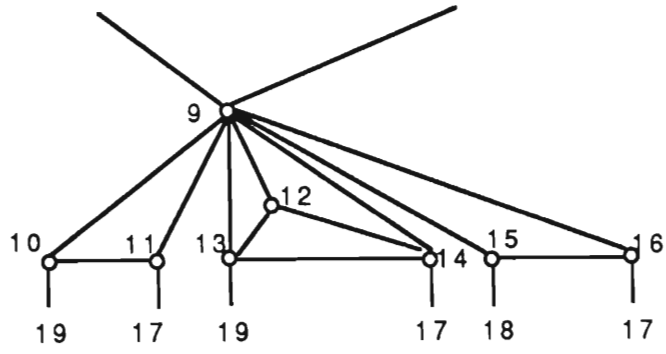
{ P-node, not pertinent root, two or more partial children }
{ see Figure 4.27 }

Let the partial children be $P_1, P_2, ..., P_k$; where $k \geq 2$
Denote the full and empty endmost children of any P_i by
Full(P_i) and Empty(P_i)
Partition the partial children into pairs

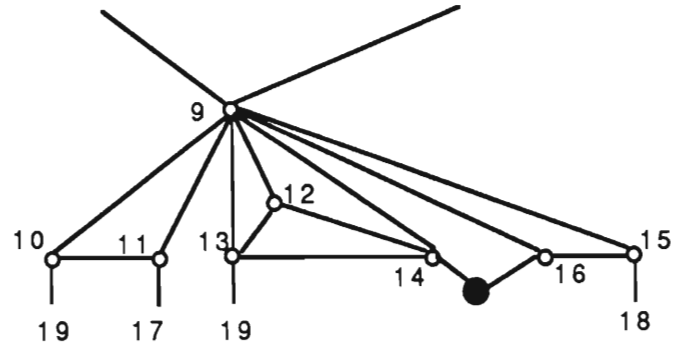
For every pair P_i and P_j do
Make Full(P_i) and Full(P_j) immediate siblings
Make new endmost child of P_i be Empty(P_j)
Delete P_j
Identify full leaves from P_i in a pseudo-vertex
$\gamma_g(G) = \gamma_g(G) + 1$
Mark full children of P_i as void
If k is odd
then

```
            Apply Template P5 to X
        else
            Apply Template P3 to X
end
```

Figure 4.27, below, illustrates the situation when Template N_4 is matched.



(a)



(b)

Figure 4.27 - Illustration of Handles when Template N_4 is matched
(a) - Before reduction for vertex 17; (b) after placement of pseudo-vertex

All that remains to be discussed is the actual placing of the second end of the handles which we insert, and to prove the algorithm correctness. The placing of the handles inserted during the reduction for a vertex i is done once the reduction process for vertex i has been completed. We have the following lemma.

Lemma 4.8: Not every sequence of pertinent children during a reduction for a vertex i is marked void.
Proof: First of all, note that, for a node X, Template N_2 and N_4 do allow X to be the pertinent root. If Template N_1 is matched, and X is the pertinent root, then Algorithm 4.15 explicitly selects a sequence $Pert_i(X)$ not to be marked void. If Template N_3 is matched, then we join two partial children, and the pertinent leaves from those two partial children, as well as the full children of X are not marked void. ❑

Using Lemma 4.8 we may identify the edges represented by full leaves not marked void in the base. Now, we place a handle from each pseudo-vertex created during the reduction for vertex i to the base, and place the corresponding identified edges along that handle.

We have the following result on the algorithm to insert handles, Algorithm 4.14.

Theorem 4.8: The Pruned Reduction Algorithm, together with Algorithm 4.14, embeds a 2-connected graph G on a surface of genus $\gamma_g(G)$, and has $O(p^3 + q)$ time complexity.
Proof: Firstly, note that, since Algorithm 4.14 is only called when the normal PQ-tree reduction fails, for a planar graph G, $\gamma_g(G) = \gamma(G) = 0$. Thus, the result follows for planar graphs from the result on the complexity of the normal PQ-tree reduction algorithm.
Next, we prove that G is correctly embedded on a surface of genus $\gamma_g(G)$. This result follows if we consider that

(i) we do not insert handles from every maximal consecutive sequence of pertinent children. This result follows directly from Lemma 4.10;
(ii) all pertinent nodes in the sequences marked void are removed from the current PQ-tree, and any pertinent nodes marked void have a handle placed at the corresponding pseudo-vertex, to allow them to be redirected to the vertex i without any edges crossing.

From point (i), each reduction pass is allowed to be completed, since we may successfully place the ends of the handles. Point (ii) ensures that the

reduction is correct, i.e. no edges cross and all edges are present. Thus, $\gamma_m(G) \geq \gamma_g(G) \geq \gamma(G)$, and the algorithm produces an embedding of G on a surface with genus $\gamma_g(G)$.

Consider now the complexity of Algorithm 4.14. The only detail that needs to be elaborated upon is the complexity of our Reduce($Pert_i(X)$) operation for all maximal subsequences, $Pert_i(X)$, of pertinent children of X. To successfully perform Reduce($Pert_i(X)$), we proceed as follows. We proceed along the sibling chains of X, from left to right. If we encounter a full or partial child, then we start a new maximal sequence $Pert_i(X)$. We continue scanning through the sibling chains until we encounter a partial or empty child. If the child is partial, then we accept it into our sequence $Pert_i(X)$, and in both situations we stop adding to our sequence $Pert_i(X)$. Then, we repeat the process on succeeding siblings, generating other maximal subsequences $Pert_j(X)$, until we encounter an endmost sibling.

From the above discussion we see that Reduce($Pert_i(X)$) has complexity $O(|\text{Children of X}|)$. Furthermore, we may scan these children O(p) times during a reduction pass. From Section 4.1, Lemma 4.1, $O(|\text{Children}(X)|) = O(p)$ as well. So, during a reduction pass for a vertex, Reduce($Pert_i(X)$) has complexity $O(p^2)$. Therefore, overall, the complexity of Reduce($Pert_i(X)$) is $O(p^3)$, because there are p reduction passes.

It is not hard to see that all other operations only consider pertinent nodes during the reduction, and so we get that Algorithm 4.14 has complexity $O(p^3 + q) = O(p^3)$. ❑

To conclude this section, we observe that the Pruned Reduction Algorithm together with Algorithm 4.14 does not necessarily produce a 2-cell embedding of G. Let H be the plane subgraph of G obtained by deleting all the edges of G that were ever marked void. Then, it may happen that we place, for two different reductions i and j, two handles from the same two regions with respect to the embedding of H. Consider Figure 4.28, below, with two edges on two handles, namely the edges e_1 = 11 17 and e_2 = 13 18.

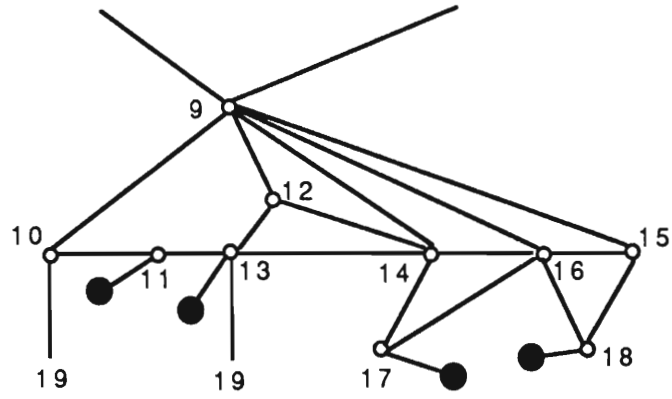


Figure 4.28 - An example of an embedding of G that is not a 2-cell

It is easy to see that the embedding of G in Figure 4.28 is not a 2-cell, if we place a closed curve C along both handles which then cannot be shrunk to a single point.

It remains an open problem to determine if Algorithm 4.14 may be extended to produce 2-cell embeddings of a non-planar graph G on some surface.

References

[AB88] S.C. Althoen and R.J. Bumcrot, *Introduction to Discrete Mathematics*, PWS-Kent, Boston (1988)

[AP61] L. Auslander and S.V. Parter, On Imbedding Graphs in the Plane, *J. Math. and Mech. 10 (3)* (1961), 517-523

[BHKY62] J. Battle, F. Harary, Y. Kodama and J.W.T. Youngs, Additivity of the genus of a graph, *Bull. Amer. Math. Soc. 68* (1962), 565-568

[BCL79] M. Behzad, G. Chartrand and L. Lesniak-Foster, *Graphs and Digraphs*, Wadsworth International, California (1979)

[BL76] K.S. Booth and G.S. Lueker, Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms, *J. Comput. Syst. Sci. 13* (1976), 335-379

[BL79] K.S. Booth and G.S. Lueker, Interval Graph Isomorphism, *J. ACM 26* (1979), 183-195

[CN88] N. Chiba and T. Nishizeki, Planar Graphs: Theory and Algorithms, *Annals of Discrete Math. (32)*, North Holland, Amsterdam (1988)

[CNAO85] N. Chiba, T. Nishizeki, S. Abe and T. Ozawa, A Linear Algorithm for Embedding Planar Graphs using PQ-trees, *J. Comput. Syst. Sci. 30* (1985), 54-76

[CNS79] N. Chiba, I. Nishioka and I. Shirakawa, An algorithm of Maximal Planarization of Graphs, *Proc. 1979 IEEE Int. Symp. on Circuits and Systems*, 919-923

[Col89] E.J. Borowski and J.M. Borwein, *Dictionary of Mathematics*, Collins Reference Series, Glasgow (1989)

[CO90] G. Chartrand and O.R. Oellermann, *Applied and Algorithmic Graph Theory*, McGraw Hill, preprint

[CON85] N. Chiba, K. Onoguchi and T. Nishizeki, Drawing plane graphs nicely, *Acta Informatica 22* (1985), 187-201

[CYN84] N. Chiba, T. Yamanouchi and T. Nishizeki, Linear Algorithms for Convex Drawings of Planar Graphs, *Progress in Graph Theory* (ed.s J.A. Bondy and U.S.R. Murty) (1984), 153-173

[DMP64] G. Demoucron, T. Malgrange and R. Pertuiset, Graphs Planaires : Reconnaisance et Construction de Représentations Plainaires Topologiques, *Revue Française de Recherche Opérationnelle 8* (1964), 34-37

[Eve79] S. Even, *Graph Algorithms*, Computer Science Press, Maryland (1979)

[ET76] S. Even and R.E. Tarjan, Computing an st-numbering, *Th. Comp. Sci. 2* (1976), 339-344

[Far46] I. Fáry, On straight line representation of planar graphs, *Acta Sci. Math. 11* (1946), 229-233

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., New York (1979)

[Gol63] A.J. Goldstein, An Efficient and Constructive Algorithm for Testing Whether a Graph can be Embedded in the Plane, *Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of of Naval Research Logistics Proj., Dept. of Math., Princeton Univ.* (1963), 2 pp

[HT72] J. Hopcroft and R. Tarjan, Depth-First Search and Linear Graph Algorithms, *SIAM J. Comput. 1* (1972), 146-160

[HT73a] J. Hopcroft and R. Tarjan, Algorithm 447: Efficient Algorithms for Graph Manipulation, *Comm. ACM 16* (1973), 372-378

[HT73b] J. Hopcroft and R. Tarjan, Dividing a Graph into triconnected components, *SIAM J. Comput. 2 (3)* (1973), 135-158

[HT74] J. Hopcroft and R. Tarjan, Efficient Planarity Testing, *J. ACM 21* (1974), 549-568

[HS76] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, California (1976)

[JTS89] R. Jayakumar, K. Thulasiraman and M.N.S. Swamy, $O(n^2)$ Algorithms for Graph Planarization, *IEEE trans. Comput.-Aided Design 8 (3)* (1989), 257-267

[JR82] L.W. Johnson and R.D. Reiss, *Numerical Analysis*, Addison-Wesley, Reading (1982)

[Kar88] A.C. Karabeg, PQ-tree data structure and some Graph Embedding Problems, Ph.D. Dissertation, University of California at San Diego (1988)

[Kar89] A.C. Karabeg, PQ-trees and Triconnectivity, presented at *Sixth British Combinatorial Conference*, London (1989)

[Knu73] D.E. Knuth, *The Art of Computer Programming, Vol. 3 Sorting and Searching*, Addison-Wesley, Reading (1973)

[Kur30] K. Kuratowski, Sur le Probléme des Courbes Gauches en Topologie, *Fund. Math. 15* (1930), 217-283

[LEC67] A. Lempel, S. Even and I. Cederbaum, An Algorithm for Planarity Testing of Graphs, *Theory of Graphs, Int. Symp., Rome (1966)*, Gordon and Breach, New York (1967)

[Mac37] S. Maclaine, A structural characterization of planar combinatorial graphs, *Duke Math. J. 3* (1937), 460-472

[Man83] A. Mansfield, Determining the Thickness of Graphs is NP-hard, *Math. Proc. Camb. Phil. Soc. 93* (1983), 9 - 23

[Meh84] K. Mehlhorn, *Data Structures and Algorithms, Vol. 2 : Graph Algorithms and NP-Completeness*, Monographs on Theoretical Computer Science, Springer Verlag, Berlin (1984)

[NSW71] E.A. Nordhaus, B.M. Stewart and A.T. White, On the maximum genus of a graph, *J. Combin. Theory 11B* (1971), 258-267

[OT81] T. Ozawa and H. Takahashi, A graph-planarization Algorithm and its application to Random Graphs, *Graph Theory and Algorithms*, Springer-Verlag Lecture Notes in Computer Science 108 (1981), 95-107

[Rea88] R.C. Read, Methods of Computer Display and Manipulation of Graphs, and the Corresponding Algorithms, *Congress. Numer. 63 (1988)*, 49-88

[Rin72] R.D. Ringeisen, Determining all compact orientable 2-manifolds upon which $K_{m,n}$ has 2-cell embeddings, *J. Combin. Theory 12B* (1972), 101-104

[Rin65] G. Ringel, Das Gesclecht des vollständigen paaren Graphen, *Abh. Math. Sem. Univ. Hamburg 28* (1965), 139-150

[RY68] G. Ringel and J.W.T. Youngs, Solution of the Heawood map-coloring problem, *Proc. Nat. Acad. Sci. USA 60* (1968), 438-445

[Sut85] G.C.J. Sutcliffe, Computerised Representation and Manipulation of Graphs, M.Sc. Thesis, University of Natal (Durban) (1985)

[TH72] R. Tarjan and J. Hopcroft, Finding the triconnected components of a graph, *Tech. Rep. 72-140, Dept. of Computer Science, Cornell University, Ithaca, New York* (1972)

[Tho80] C. Thomassen, Planarity and Duality of finite and infinite Graphs, *J. Combin. Theory Series B29* (1980), 244-271

[TJS86] K. Thulasiraman, R. Jayakumar and M.N.S. Swamy, On Maximal Planarization of Nonplanar Graphs, *IEEE trans. Circuits and Systems CAS-33, 8* (1986), 843-844

[Tut60] W.T. Tutte, Convex Representation of Graphs, *Proc. London Math. Soc. 10 (3)* (1960), 304-320

[Tut63] W.T. Tutte, How to Draw a Graph, *Proc. London Math. Soc. Ser. 3, 13* (1963), 743-768

[Wil84] S.G. Williamson, Depth-First Search and Kuratowski Subgraphs, *J. ACM 31, no 4* (1984), 681-693

[Wil85] S.G. Williamson, *Combinatorics for Computer Science*, Computer Science Press, Maryland (1985)

[You63] J.W.T. Youngs, Minimal imbeddings and the genus of a graph, *J. Math. Mech. 12* (1963), 303-315

Appendix A

Source Code Listings

```
program Planarity_Testing_Algorithms;

{-------------------------------------------------------------}
{-             The Driver Program                            -}
{-------------------------------------------------------------}

uses
  Planar_Defs,          { Graph and type definitions }
  Planar_Miscellaneous, { Global Routines used by all the algorithms }
                        { The Algorithm units :- }
  Hop_Algorithm,            { Hopcroft and Tarjan }
  DMP_Algorithm,
  Lempel_Algorithm,               { Lempel, Even and Cederbaum }
  CRT;                  { System unit for Screen control }


{-------------------------------------------------------------}
{- We Print the actual information to do with the Demo.      -}
{-------------------------------------------------------------}
procedure PrintInfo;

begin
  TextBackground(Black);
  Clrscr;
  TextColor(Red);
  Writeln ('Welcome to the first Project Demonstration Program');
  Writeln;
  Writeln ('                The Planarity Testing Program');
  Writeln;
  Writeln;
  Writeln ('The Demonstration allows you to select a test file from a list.');
  Writeln ('You may then select between the three available planarity testing');
  Writeln ('Algorithms.');
  Writeln;
  Writeln ('The demonstration will pause after each stage to allow you to ');
  Writeln ('study the output. Please press ''Q'' to exit from the demonstration.');
  Writeln;
  Writeln;
  Writeln;
  Writeln;
  Prompt
end;

{-------------------------------------------------------------}
{- Exit Message to end tidly.                                -}
{-------------------------------------------------------------}
procedure Say_Bye;

begin
  GotoXY (30, 10);
  TextColor(Yellow);
  Writeln ('Returning to the Batch File');
  TextColor (Black); Write (' ');
  Delay (300)
end;
```

```
{-------------------------------------------------------}
{- Displays the menu and accepts user's choice.        -}
{-------------------------------------------------------}
procedure Menu;

var
  Heap_Posn : ^integer;
  Finished : Boolean;
  Choice   : Char;

procedure Test_Planarity;

var
  Choice : Char;

begin
  Clrscr;
  TextColor (Red);
  writeln ('Please Choose Method of Planarity Testing');
  TextColor (Yellow);
  writeln;
  writeln (' ':8, '1  : Lempel, Even and Cederbaum');
  writeln (' ':8, '2  : Hopcroft and Tarjan');
  writeln (' ':8, '3  : Democron, Malgrange and Pertuiset');
  writeln (' ':8, 'Q  : Quit this Demonstration');
  writeln;
  TextColor(Red);
  write (' ':8, '  Please enter your choice [ ]', #8#8);
  TextColor (Yellow);
  Choice := UpCase (ReadKey);
  if Choice >= #32
    then write (Choice, #8)
    else write (' ', #8);
  case Choice of
    '1' : if File_Loaded
              then Lempel_Even_Cederbaum
              else begin
                writeln;
                writeln ('No File Loaded');
                prompt;
              end;
    '2' : if File_Loaded
            then Hopcroft_Tarjan
            else begin
              writeln;
              writeln ('No File Loaded');
              prompt;
            end;
    '3' : if File_Loaded
            then Demoucran_Malgrange_Pertuisel
            else begin
              writeln;
              writeln ('No File Loaded');
              prompt;
            end;
    #27, 'Q' : Finished := true
  end
end;
```

```
{-------------------------------------------------------}
{- Main menu of procedure.                             -}
{-------------------------------------------------------}
begin
  Finished := False;
  repeat
    TextBackground (Black);
    ClrScr;
    Mark (Heap_Posn);          { note memory position }
    Initialise_Graph;
    Build_Graph;
    if File_Loaded
      then Test_Planarity;
    Release (Heap_Posn);       { release memory back to old position }
    writeln;
    writeln;
    while Keypressed do
      Choice := readkey;
    if not Finished
      then begin
        TextColor (Red);
        writeln ('Press any key to continue or ''Q'' to Quit this demonstration');
         TextColor (Yellow);
        Choice := upcase(Readkey);
        Finished := (Choice = 'Q')
      end
  until Finished;
  Writeln;
  TextColor (Red);
  writeln ('Press ''Y'' to test your own graph, or any other key to continue');
  TextColor (Yellow);
  Choice := upcase(Readkey);
  if Choice = 'Y'
    then begin
      Enter_Own_Graph;
      if File_Loaded
        then Test_Planarity
        else begin
          Writeln;
          Writeln ('Sorry - No Graph was accepted for input.');
        end;
      writeln;
      prompt
    end;
  TextColor(LightGray);
  Clrscr
end;
```

```
{-------------------------------------------------------------}
{- Main Program.                                             -}
{-------------------------------------------------------------}
begin
  PrintInfo;
  File_Loaded := false;
  Initialise_Graph;
  Menu;
  Say_Bye
end.
```

```
{-------------------------------------------------------}
{- The Lempel, Even and Cederbaum Algorithm             -}
{-------------------------------------------------------}
Unit Lempel_Algorithm;

interface

procedure Lempel_Even_CederBaum;

implementation

uses
  Planar_Defs,
  Planar_Miscellaneous,
  Lempel_Globals,          { Global Routines used by Reduction and Bubble phases }
  Lempel_Reduce,           { Reduction unit                                      }
  Lempel_Bubble,           { Bubble unit                                         }
  CRT;

{-------------------------------------------------------}
{- Main program                                         -}
{-------------------------------------------------------}
procedure Lempel_Even_Cederbaum;

{-------------------------------------------------------}
{- The Depth-First routine is as per Hopcroft and Tarjan, -}
{- but we do not need to compute L2.                    -}
{-                                                      -}
{- See Chapter 1                                        -}
{-------------------------------------------------------}
procedure DFS;

function Min (x, y : Integer) : Integer;

begin
  if x < y
    then Min := x
    else Min := y;
end;
```

```
var
  Current_Label : 0..MaxInt;
  Temp_Edge     : Edge_Ptr;
  Finished,
  New_Vertex    : Boolean;
  Temp_Vertex   : Vertex_Ptr;

{-------------------------------------------------------------}
{- The main DFS procedure                                     -}
{-------------------------------------------------------------}
begin
{Initialising all Fathers, Labels to 0 }
  For Temp_Vertex := 1 to Last_Vertex do
    with Graph^ [Temp_Vertex] do
      begin
        Number := 0;
        Father := 0;
        L1 := 0;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do
          begin
            Temp_Edge^. Used := False;
            Temp_Edge^. Deleted := False;
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;
  New_Vertex := true;
  Temp_Vertex := 1;
  Current_Label := 0;
  repeat
    repeat
      repeat
        {2}
        if New_Vertex
          then begin
            Current_Label := Current_Label + 1;
            Graph^ [Temp_Vertex]. Number := Current_Label;
            Graph^ [Temp_Vertex]. L1 := Current_Label;
          end;
        {3}
        Temp_Edge := Graph^ [Temp_Vertex]. Edges;
        While (Temp_Edge <> Nil) and ((Temp_Edge^. Used) or (Temp_Edge^. Deleted)) do
          Temp_Edge := Temp_Edge^. Next;     { search for unused edge }
        if (Temp_Edge <> Nil)
          then begin
            {4}
              { Direct the Edge }
              Temp_Edge^. Used := True;
              Temp_Edge^. Other_Edge^. Deleted := true;
              with Graph^ [Temp_Edge^. Vertex] do
              if Number <> 0
                then begin { Back Edge - adjust L1 and return }
                  if Number < Graph^ [Temp_Vertex]. L1
                    then begin
                      Graph^ [Temp_Vertex]. L1 := Number;
                    end
                    else;
                  New_Vertex := false;
```

```
            end
            else begin
              Father := Temp_Vertex;
              Temp_Vertex := Temp_Edge^. Vertex;
              New_Vertex := true;
            end;
        end
    until (Temp_Edge = Nil);              { until we have to backtrack }
    if Graph^ [Temp_Vertex]. Number = 1
      then
        {5} Finished := true
      else begin
        {6}
        with Graph^ [Graph^ [Temp_Vertex]. Father] do    { update L1 }
          if Graph^ [Temp_Vertex]. L1 < L1
            then L1 := Graph^ [Temp_Vertex]. L1;
        Temp_Vertex := Graph^ [Temp_Vertex]. Father;
        New_Vertex := false;
      end
  until Finished;      { until explored entire graph }
end;

{-------------------------------------------------------}
{- We reorder the lists to ensure that we always choose an  -}
{- edge with the L1 weighting first.                        -}
{- The algorithm is as per Hopcroft and Tarjan.             -}
{-------------------------------------------------------}
procedure ReOrder_Lists;

{-------------------------------------------------------}
{- Simplified weighting - only L1 necessary.                -}
{-------------------------------------------------------}
function Phi (u, v : Vertex_Ptr) : Integer;

begin
  if Graph^ [v]. Number < Graph^ [u]. Number
    then Phi := Graph^ [v]. Number
    else Phi := Graph^ [v]. L1
end;
```

```
var
  Bucket_Array : array [1..2 * Max_Vertices + 1] of Bucket_Ptr;
  Temp_Bucket,
  Temp_Bucket2 : Bucket_Ptr;
  Temp_Vertex  : Integer;
  Temp_Edge    : Edge_Ptr;

{-------------------------------------------------------------}
{- The main Reorder_Lists procedure                           -}
{-------------------------------------------------------------}
begin
  For Temp_Vertex := 1 to 2 * Last_Vertex + 1 do
    Bucket_Array [Temp_Vertex] := Nil;
  For Temp_Vertex := 1 to Last_Vertex do
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while (Temp_Edge <> Nil) do
        begin
          Temp_Edge^. Weight := Phi (Temp_Vertex, Temp_Edge^. Vertex);
          new (Temp_Bucket);
          Temp_Bucket^. Data := Temp_Edge;
          Temp_Bucket^. Next := Bucket_Array [Temp_Edge^. Weight];
          Temp_Bucket^. Vertex := Temp_Vertex;
          Bucket_Array [Temp_Edge^. Weight] := Temp_Bucket;
          Temp_Edge := Temp_Edge^. Next
        end;
      Graph^ [Temp_Vertex]. Edges := Nil;
    end;
  For Temp_Vertex := 2 * Last_Vertex + 1 downto 1 do
    begin
      Temp_Bucket := Bucket_Array [Temp_Vertex];
      while Temp_Bucket <> Nil do
        begin
          Temp_Bucket^. Data^. Next := Graph^ [Temp_Bucket^. Vertex]. Edges;
          Graph^ [Temp_Bucket^. Vertex]. Edges := Temp_Bucket^. Data;
          Temp_Bucket2 := Temp_Bucket;
          Temp_Bucket := Temp_Bucket^. Next;
          Dispose (Temp_Bucket2);
        end;
    end;
end;
```

```
{-------------------------------------------------------}
{- The main initialise for LEC algorithm               -}
{-------------------------------------------------------}
procedure Global_Initialise_Lempel;

const
  New_Mark = 1;   { unused vertex mark }
  Old_Mark = 0;   { used vertex mark   }

var
  Temp_Edge : Edge_Ptr;       { temporary variable }
  Loop      : Vertex_Ptr;     { temporary variable }

begin
  DFS;                                  { for the ST numbering and for 2-Connected components }
  if Check_2_Connected
    then begin
      ReOrder_Lists;                      { to ensure we choose correct paths }
                                          { are chosen during ST-Numbering    }
      for Loop := 1 to Max_Vertices do    { reset the vertices and }
        begin                             { edges to initial value for ST Numbering }
          Graph^ [Loop]. Mark := New_Mark;
          Graph^ [Loop]. Used := false;   { we have not tested this component yet }
          Temp_Edge := Graph^ [Loop]. Edges;
          while Temp_Edge <> Nil do
            begin
              Temp_Edge^. Mark := New_Mark;
              Temp_Edge := Temp_Edge^. Next;
            end;
        end;
      Planar := true
    end
    else Planar := false
end;
```

```
{-------------------------------------------------------}
{- St-Numbering for the graph is computed. See chapter 1 for-}
{- details on the algorithm.                            -}
{-------------------------------------------------------}
procedure ST_Numbering (Start_Vertex : Vertex_Ptr);

const
  New_Mark = 1;   { unused vertex mark }
  Old_Mark = 0;   { used vertex mark   }

var
  ST_Stack     : Array [1..Max_Vertices] of Vertex_Ptr; { stack of vertices to give numbers to }
  Top_of_Stack : 0..Max_Vertices;                       { top of the above stack              }
  Current_Number,                  { current ST number to assign }
  Loop         : Integer;          { temporaray variable         }
  Current_Vertex : Vertex_Ptr;     { Current vertex the algorithm is at }
  Temp_Edge      : Edge_Ptr;       { temporary variable }

{-------------------------------------------------------}
{- We are adding a path from Current vertex down the tree   -}
{- until we reach a back edge.                              -}
{- The routine is necessarily recursive, since we need to   -}
{- add the path in reverse order.                           -}
{-------------------------------------------------------}
procedure Add_Forward_Path_to_Stack (Current_Edge : Edge_Ptr);

var Temp_Edge : Edge_Ptr;

begin
  if Graph^ [Current_Edge^. Vertex]. Mark <> Old_Mark   { see if we stop yet }
    then begin
      Temp_Edge := Graph^ [Current_Edge^. Vertex]. Edges;  { get first valid edge }
      while Temp_Edge^. Deleted do
        Temp_Edge := Temp_Edge^. Next;
      Add_Forward_Path_to_Stack (Temp_Edge);              { add path from there to the path }
      Current_Edge^. Mark := Old_Mark;                    { Note that the edge is used      }
      Top_of_Stack := Top_of_Stack + 1;
      ST_Stack [Top_of_Stack] := Current_Edge^. Vertex;   { and add this vertex at the TOS  }
      Graph^ [Current_Edge^. Vertex]. Mark := Old_Mark    { Note that the vertex is stacked }
    end
    else Current_Edge^. Mark := Old_Mark                  { At the end of the path          }
end;
```

```
{-------------------------------------------------------}
{- We are adding a path from the vertex in reverse direction-}
{- back along a directed path.                          -}
{- The routine is necessarily recursive, since we need to   -}
{- add the path in reverse order.                       -}
{-------------------------------------------------------}
procedure Add_Backward_Path_to_Stack (Current_Edge : Edge_Ptr);

var Temp_Edge : Edge_Ptr;

begin
  if Graph^ [Current_Edge^. Vertex]. Mark <> Old_Mark   { see if the path is finished yet }
    then begin
      Temp_Edge := Graph^ [Current_Edge^. Vertex]. Edges;
      while (Temp_Edge^. Vertex <> Graph^ [Current_Edge^. Vertex]. Father) do
        Temp_Edge := Temp_Edge^. Next;                  { get a valid edge }
      Add_Backward_Path_to_Stack (Temp_Edge);           { add the path to the stack }
      Top_of_Stack := Top_of_Stack + 1;
      ST_Stack [Top_of_Stack] := Current_Edge^. Vertex; { add this vertex }
      Current_Edge^. Mark := Old_Mark;                  { note the edge is used }
      Graph^ [Current_Edge^. Vertex]. Mark := Old_Mark  { and the vertex is on the stack }
    end
    else Current_Edge^. Mark := Old_Mark                { at the end of the path }
end;

{-------------------------------------------------------}
{- The main ST-Numbering routine                        -}
{-------------------------------------------------------}
begin
  Top_of_Stack := 2;    { Elements 1 and n only             }
  ST_Stack [1] := Start_Vertex;               { Vertex 1 is S }
  ST_Stack [2] := Graph^ [Start_Vertex]. Edges^. Vertex; { and this is T }
  Current_Number := 1;
  Graph^ [1]. Mark := Old_Mark;
  Temp_Edge := Graph^ [Start_Vertex]. Edges;
  Graph^ [Temp_Edge^. Vertex]. Mark := Old_Mark;
  Temp_Edge^. Mark := Old_Mark;
  Temp_Edge^. Other_Edge^. Mark := Old_Mark;  { Note the edge between them is used }
  while Top_of_Stack > 0 do                   { While a number needs an element }
    begin
      if Top_of_Stack = 1      { ending condition - last vertex S is on the stack }
        then begin
          Graph^ [ST_Stack [1]]. St_Number := Current_Number;  { Assign the last ST-number }
          Graph^ [ST_Stack [1]]. Used := true;                 { Note it has been tested   }
          ST_Number_Index [Current_Number] := ST_Stack [1];    { to the last vertex        }
          Top_of_Stack := 0                                    { we are finished                }
        end
        else begin
          Current_Vertex := ST_Stack [Top_of_Stack];    { get the vertex }
          Top_of_Stack := Top_of_Stack - 1;
          Temp_Edge := Graph^ [Current_Vertex]. Edges;
          while (Temp_Edge <> Nil) and (Temp_Edge^. Mark <> New_Mark) do
            Temp_Edge := Temp_Edge^. Next;         { get first unused edge }
          if Temp_Edge = Nil                       { if none left          }
            then begin
              Graph^ [Current_Vertex]. ST_Number := Current_Number;  { Remove from stack i.e. don't add it ba
              Graph^ [Current_Vertex]. Used := true;                 { Note it has been tested   }
              ST_Number_Index [Current_Number] := Current_Vertex;    { Note the vertex assigned this number }
```

```
            Graph^ [Current_Vertex]. Mark := Old_Mark;          { give an ST - number to the vertex    }
            Current_Number := Current_Number + 1;               { and move to next numbering available }
          end
          else
            if Graph^ [Temp_Edge^. Vertex]. Number
                 < Graph^ [Current_Vertex]. Number        { check if it is a back edge }
              then begin
                Temp_Edge^. Mark := Old_Mark;            { note the edge used }
                Top_of_Stack := Top_of_Stack + 1;        { and re-add the vertex to the stack }
                ST_Stack [Top_of_Stack] := Current_Vertex
              end
              else
                if not Temp_Edge^. Deleted               { if the edge is a tree edge }
                  then begin
                    Add_Forward_Path_to_Stack (Temp_Edge);    { add the path to the stack       }
                    Top_of_Stack := Top_of_Stack + 1;         { in order so that current vertex }
                    ST_Stack [Top_of_Stack] := Current_Vertex { is on top of the stack          }
                  end
                  else begin                             { if the edge is a back edge }
                                                         { ending at this vertex }
                    Add_Backward_Path_to_Stack (Temp_Edge);   { add the path to the stack       }
                    Top_of_Stack := Top_of_Stack + 1;         { in order so that current vertex }
                    ST_Stack [Top_of_Stack] := Current_Vertex { is on top of the stack          }
                  end
      end
    end
end;

{-------------------------------------------------------------}
{- Main initialisation Routine for PQ-trees                  -}
{-------------------------------------------------------------}
procedure Initialise_LEC (Start_Vertex : Vertex_Ptr);

var
  Loop    : Vertex_Ptr;
  Temp_PQ : PQ_Node_Ptr;

begin
  ST_Numbering (Start_Vertex);      { Generate the ST-Numbering                  }
  Used_List := Nil;                 { Used List is for Tree node reinitialisation }
                                    { during the program's running.              }
  Create_PQ_Node (Temp_PQ);         { Build the root }
  Temp_PQ^. Node_Type := P_Node;
  Temp_PQ^. Child_Count := 0;
  Temp_PQ^. List_Start := Nil;
  Start_Vertex := ST_Number_Index [1];          { Note the starting vertex }
  Add_Edges_to_Tree (Start_Vertex, Temp_PQ);    { and add the leaves       }
  Sizeof_Queue := 0;                            { Nothing on the Queue         }
  Num_Vertices_Added := 1;                      { We have only added the root }
end;
```

```
{-------------------------------------------------------}
{- The nodes affected during this pass of the algorithm will-}
{- be reset to initial values.                          -}
{-------------------------------------------------------}
procedure Reset_Marked_Vertices;

var
  List_Element,
  List_Element2 : List_Ptr;

begin { Reset Marked Vertices }
  if Pseudo_Node <> Nil
    then dispose (Pseudo_Node);    { Kill the pseudo node }
  List_Element := Used_List;
  while List_Element <> Nil do     { For every used element }
    begin
      if List_Element^. Element <> Pseudo_Node
        then
          if List_Element^. Element^. Data_Label <> Full   { If it was not Full }
            then with List_Element^. Element^ do
                   begin                          { Reset all the relevant fields }
                     Full_Kids := Nil;
                     Partial_Kids := Nil;
                     Full_Kids_Count := 0;
                     Data_Label := Empty;
                     Mark := None;
                     Pert_Leaf_Count := 0;
                     Pert_Child_Count := 0
                   end
            else dispose (List_Element^. Element);    { For a full node get memory }
      List_Element2 := List_Element;
      List_Element := List_Element^. Next;         { go to next element }
      dispose (List_Element2)
    end;
  Used_List := Nil                                 { Note no new elements }
end;
```

```
{-------------------------------------------------------}
{- The full kids are removed and as per vertex addition in  -}
{- main algorithm, we insert a new P-node in their place.   -}
{- This routine is called only for Q-node parents.          -}
{-------------------------------------------------------}
procedure Remove_Full_Kids_Insert_new_Node (Current_Node,
                                            New_Node,
                                            Start_Node_Sib,
                                            End_Node_Sib    : PQ_Node_Ptr);
var
  New_Indicator,
  Temp_Node,
  Full_Sib,
  Empty_Sib,
  Full_Kid,
  Empty_Kid,
  Left_Kid,
  Right_Kid  : PQ_Node_Ptr;
  Temp_List  : List_Ptr;

begin
  Left_Kid := Nil;
  Right_Kid := Nil;
  Temp_List := Current_Node^. Full_Kids;      { find the endmost kids }
  while Right_Kid = Nil do
    begin
      Temp_Node := Nbour_of (Temp_List^. Element);
      if Temp_Node^. Data_Label <> Full
        then if Left_Kid = Nil
               then Left_Kid := Temp_List^. Element
               else Right_Kid := Temp_List^. Element;
      Temp_Node := Other_NBour_of (Temp_List^. Element);
      if (Temp_Node = Nil) or (Temp_Node^. Data_Label <> Full)
        then if Left_Kid = Nil
               then Left_Kid := Temp_List^. Element
               else Right_Kid := Temp_List^. Element;
      Temp_List := Temp_List^. Next
    end;
  Get_Full_Empty_Siblings (Right_Kid, Full_Sib, Empty_Sib);   { Get the neighbours }
  if Right_Kid = Left_Kid
    then begin
      if (Empty_Sib <> Nil)                                   { and replace the sequence }
        then Add_Replace_Sibling (New_Node, Right_Kid, Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node);
      if (Full_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Right_Kid, Full_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node)
    end
    else begin
      if (Empty_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Right_Kid, Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node);
      Get_Full_Empty_Siblings (Left_Kid, Full_Sib,
                               Empty_Sib);   { Get the neighbours }
      if (Empty_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Left_Kid, Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Left_Kid, New_Node)
    end
```

```
end;


{-------------------------------------------------------------}
{- Special case where Root of pertinent subtree is a Pseudo -}
{- Node.                                                    -}
{- The case is a simplification to the normal, since we do  -}
{- not have to check for endmost children.                  -}
{-------------------------------------------------------------}
procedure Add_New_Q_Root_Pseudo_Node (var New_Root : PQ_Node_Ptr);

var
  Finished   : Boolean;
  Start_Node_Sib,
  End_Node_Sib,
  Full_Sib,
  Empty_Sib,
  Temp_Node,
  Temp_Node2 : PQ_Node_Ptr;
  Temp_List  : List_Ptr;

begin
  Temp_List := Root_Node^. Full_Kids;       { get an full kid adjacent to an empty kid }
  Finished := false;
  while not Finished do
    begin
      if (NBour_of (Temp_List^. Element)^. Data_Label <> Full)
       or (Other_NBour_of (Temp_List^. Element)^. Data_Label <> Full)
        then Finished := true
        else Temp_List := Temp_List^. Next
    end;
  Temp_Node := Temp_List^. Element;
  Get_Full_Empty_Siblings (Temp_Node, Full_Sib, Empty_Sib);
  Start_Node_Sib := Empty_Sib;
  Temp_Node2 := Full_Sib;
  while (Temp_Node2 <> Nil)                          { traverse sequence to end }
    and (Temp_Node2^. Data_Label = Full) do
      Walk_Normal (Temp_Node, Temp_Node2);
  End_Node_Sib := Temp_Node2;

  Create_PQ_Node (New_Root);    { New node to insert }
  with New_Root^ do
    begin
      Node_Type := P_Node;
      List_Start := Nil;
      Child_Count := 0
    end;
  Remove_Full_Kids_Insert_new_Node (Pseudo_Node, New_Root,
                                    Start_Node_Sib, End_Node_Sib);
  Add_List_to_Used_List (Root_Node^. Full_Kids);            { Wipe the kid }
  Add_Node_to_Used_List (Root_Node);                        { and wipe the Pseudo node }
end;
```

```
{-------------------------------------------------------}
{- An ordinary Q-root replacement.                      -}
{-------------------------------------------------------}
procedure Add_New_Q_Root (var New_Root : PQ_Node_Ptr);

var
  List_Element : List_Ptr;
  Start_Node_Sib,
  End_Node_Sib,
  Temp_Node2,
  Temp_Node,
  Full_Sib,
  Empty_Sib,
  Sibling1,
  Sibling2     : PQ_Node_Ptr;
  Finished     : Boolean;

begin
  List_Element := Root_Node^. Full_Kids;
  Finished := false;
  while not Finished do                        { get an endmost kid }
    begin
      Temp_Node := NBour_of (List_Element^. Element);
      Temp_Node2 := Other_NBour_of (List_Element^. Element);
      if (Temp_Node^. Data_Label <> Full) or (Temp_Node2 = Nil)
          or (Temp_Node2^. Data_Label <> Full)
        then Finished := true
        else List_Element := List_Element^. Next
    end;
  Temp_Node := List_Element^. Element;
  Get_Full_Empty_Siblings (Temp_Node, Full_Sib, Empty_Sib);
  Start_Node_Sib := Empty_Sib;
  Temp_Node2 := Full_Sib;
  while (Temp_Node2 <> Nil)                    { and find other end of sequence }
    and (Temp_Node2^. Data_Label = Full) do
      Walk_Normal (Temp_Node, Temp_Node2);
  End_Node_Sib := Temp_Node2;

  Create_PQ_Node (New_Root);    { New node to insert }
  with New_Root^ do
    begin
      Node_Type := P_Node;
      Parent := Root_Node;
      List_Start := Nil;
      Child_Count := 0
    end;
  Remove_Full_Kids_Insert_new_Node (Root_Node, New_Root,      { insert it! }
                                    Start_Node_Sib, End_Node_Sib);
  Root_Node^. Data_Label := Empty;
  Add_List_to_Used_List (Root_Node^. Full_Kids);  { Reset values }
  Add_Node_to_Used_List (Root_Node);
  Temp_Node := Root_Node^. Parent;                { And add all the parents }
  while Temp_Node <> Nil do                       { to be reset as well     }
    begin
      Add_Node_To_Used_List (Temp_Node);
      Temp_Node := Temp_Node^. Parent
    end
end;
```

```
{-------------------------------------------------------}
{- Straight forward replacement of the P node with new root -}
{-------------------------------------------------------}
procedure Add_New_P_Root (var New_Root : PQ_Node_Ptr);

var
  Temp_Node    : PQ_Node_Ptr;
  Temp_Double2,
  Temp_Double  : Double_Ptr;

begin
  if Root_Node^. List_Start <> Nil                { wipe all the children }
    then begin
      Temp_Double := Root_Node^. List_Start^. Right;
      while Temp_Double <> Root_Node^. List_Start do
        begin
          Temp_Double2 := Temp_Double;
          Temp_Double := Temp_Double^. Right;
          dispose (Temp_Double2)
        end;
      dispose (Root_Node^. List_Start);
      Root_Node^. List_Start := Nil
    end;
  Root_Node^. Data_Label := Empty;
  Root_Node^. Child_Count := 0;
  Add_List_to_Used_List (Root_Node^. Full_Kids);   { Reset the values }
  Add_Node_to_Used_List (Root_Node);
  Temp_Node := Root_Node^. Parent;                 { and reset the values of the parents }
  while Temp_Node <> Nil do
    begin
      Add_Node_To_Used_List (Temp_Node);
      Temp_Node := Temp_Node^. Parent
    end;
  New_Root := Root_Node
end;
```

```
begin
  Global_Initialise_Lempel;              { Do main initialisation }
  if Planar
    then begin
      Current_Vertex := 1;
      Initialise_LEC (Current_Vertex);                    { Initialise a tree and ST-Numbering }
      while (Num_Vertices_Added <= Last_Vertex - 1) and (Planar) do
        begin
          Num_Vertices_Added := Num_Vertices_Added + 1;
          writeln ('Bunching ST number ', Num_vertices_Added, ' now...');
          Bubble_Tree;                   { Do the Bubble - Pass I     }
          Reduce_Pertinent_Subtree;      { Do the Reduction - Pass II }
          if Planar                      { Replace sub-tree from Root (T, S) }
            then begin
              if Root_Node^. Node_Type = Q_Node
                then if Root_Node = Pseudo_Node      { Two different cases }
                       then Add_New_Q_Root_Pseudo_Node (New_Root)
                       else Add_New_Q_Root (New_Root)
                else begin
                  if Root_Node^. Node_Type = Leaf    { leaf is a special case of P-node }
                    then begin
                      with Root_Node^ do
                        begin
                          Node_Type := P_Node;
                          List_Start := Nil;
                          Child_Count := 0
                        end
                    end;
                  Add_New_P_Root (New_Root)          { and replace P-node with new P-node and edges }
                end;
              Reset_Marked_Vertices;       { Reset the nodes affected during bubbling }
              if Num_Vertices_Added <> Last_Vertex
                then Add_Edges_to_Tree (ST_Number_Index [Num_Vertices_Added],
                                        New_Root);   { and add the new leaves }
            end;
        end;
      if not Planar
        then writeln ('The graph is non-planar')
        else writeln ('The graph is planar')
    end;
end;

end.
```

```
{-------------------------------------------------------------}
{- This unit contains the Bubble phase (Pass I) of the       -}
{- Reduction algorithm.                                      -}
{-------------------------------------------------------------}
unit Lempel_Bubble;

interface

procedure Bubble_Tree;        { single procedure to perform the bubble phase }

implementation

uses
  Lempel_Globals,
  Planar_Defs;

{-------------------------------------------------------------}
{- Perform the bubble.                                       -}
{-------------------------------------------------------------}
procedure Bubble_Tree;

var
  Finished           : Boolean;      { Finished the bubble }
  Blocked_List       : List_Ptr;     { List of all nodes which are blocked }
  Block_Count,                       { Blocks of blocked nodes }
  Blocked_Nodes      : Integer;      { Total number of blocked nodes }
  Loop,
  Off_the_Top        : 0..1;         { if we have reached the Root of the tree }
  Current_Parent,
  Current_Node       : PQ_Node_Ptr;
  Blocked_Siblings   : Integer;      { Number of Blocked Siblings   }

{-------------------------------------------------------------}
{- Variables are initialised to their default values.        -}
{-------------------------------------------------------------}
procedure Initialise_Bubble;

begin
  Blocked_List := Nil;
  Pseudo_Node := Nil;
  Queue_Start := Nil;
  Queue_Head := Nil;
  Block_Count := 0;
  Blocked_Nodes := 0;
  Off_the_Top := 0;
end;
```

```
{-------------------------------------------------------}
{- A Blocked node is added to the Blocked list.        -}
{-------------------------------------------------------}
procedure Add_to_Blocked_List (Node : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Element := Node;
  List_Element^. Next := Blocked_List;
  Blocked_List := List_Element
end;

{-------------------------------------------------------}
{- An Blocked node is now unBlocked, and must be removed -}
{- from the Blocked List.                              -}
{-------------------------------------------------------}
procedure Remove_from_Blocked_List (Node : PQ_Node_Ptr);

var
  Temp_Element,
  List_Element : List_Ptr;

begin
  if Blocked_List <> Nil
    then begin
      List_Element := Blocked_List;
      if List_Element^. Element = Node  { check if the element is at the Head of the list }
        then begin
          Blocked_List := Blocked_List^. Next;
          dispose (List_Element)
        end
        else begin
          while List_Element^. Next^. Element <> Node do    { search for the element }
            List_Element := List_Element^. Next;
          Temp_Element := List_Element^. Next;
          List_Element^. Next := Temp_Element^. Next;       { delete from the list }
          dispose (Temp_Element)
        end
    end
    else Halt                  { error - trying to remove from empty list }
end;
```

```
{-------------------------------------------------------}
{- The Bubble procedure has finished with Block_Count = 1.  -}
{- This means that the pertinent children of the Root are   -}
{- blocked and are in one group. So we give them a 'new'    -}
{- parent which we will delete at the end of the Reduction. -}
{- Hence Pseudo since it is not really their parent.        -}
{-------------------------------------------------------}
procedure Create_Pseudo_Node;

var
  Number_EndMost_Kids : Integer;    { We need the two end kids }
  List_Element        : List_Ptr;

begin
  Create_PQ_Node (Pseudo_Node);     { Create the new node. }
  With Pseudo_Node^ do
    begin
      Node_Type := Q_Node;
      Number_EndMost_Kids := 0;
      while (Blocked_List <> Nil) do     { set the endmost kids }
        begin
          List_Element := Blocked_List;
          with List_Element^. Element^. Immediate_Siblings^ do   { an endmost kid is defined }
            if (Element^. Mark <> Blocked)                       { as one who does not have  }
              or (Next^. Element^. Mark <> Blocked)              { both siblings Blocked.    }
              then begin
                if Number_EndMost_Kids = 1              { set the relevant endmost pointer }
                  then RightMost_Kid := List_Element^. Element
                  else LeftMost_Kid := List_Element^. Next^. Element;
                Number_EndMost_Kids := Number_EndMost_Kids + 1
              end;
          Pert_Child_Count := Pert_Child_Count + 1;        { Count the number of pertinent kids }
          List_Element^. Element^. Parent := Pseudo_Node;  { and set the parent pointer }
          Blocked_List := Blocked_List^. Next;
          dispose (List_Element)
        end;
    end
end;
```

```
{-------------------------------------------------------}
{- We determine status of the node - Blocked or UnBlocked   -}
{-------------------------------------------------------}
procedure Block_or_Unblock_Node;

var
  UnBlocked_Siblings,                { Number of Unblocked Siblings   }
  Dud_Siblings       : Integer;      { Number of neither Blocked nor UnBlocked }
  Temp_Node          : PQ_Node_Ptr;  { Temporary variable }

begin
  Current_Node^. Mark := Blocked;   { assume it is blocked }
  UnBlocked_Siblings := 0;          { Reset counts of neighbours }
  Dud_Siblings := 0;
  Blocked_Siblings := 0;
  with Current_Node^ do
    begin
      if Immediate_Siblings <> Nil
        then begin
          Temp_Node := Nbour_of (Current_Node);
          if Temp_Node^. Mark = Blocked
            then Blocked_Siblings := Blocked_Siblings + 1
            else
              if Temp_Node^. Mark = UnBlocked
                then UnBlocked_Siblings := UnBlocked_Siblings + 1
                else Dud_Siblings := Dud_Siblings + 1;  { neither blocked nor unblocked }
          Temp_Node := Other_Nbour_of (Current_Node);
          if Temp_Node <> Nil
            then
              if Temp_Node^. Mark = Blocked
                then Blocked_Siblings := Blocked_Siblings + 1
                else
                  if Temp_Node^. Mark = UnBlocked
                    then UnBlocked_Siblings := UnBlocked_Siblings + 1
                    else Dud_Siblings := Dud_Siblings + 1;  { neither blocked nor unblocked }
        end;
      if UnBlocked_Siblings <> 0        { Check if we may unblock the Node }
        then begin
          Temp_Node := NBour_of (Current_Node);
          if Temp_Node^. Mark <> UnBlocked
            then Temp_Node := Other_NBour_of (Current_Node);
          Parent := Temp_Node^. Parent;      { Give it a proper Parent Reference }
          Mark := UnBlocked
        end
        else
          if (UnBlocked_Siblings + Blocked_Siblings + Dud_Siblings < 2)   { if Parent is P Node }
            then Mark := UnBlocked;                                       { or endmost kid of parent }
    end
end;
```

```
{-------------------------------------------------------}
{- The adjacent block of Blocked Nodes may be unBlocked.    -}
{-------------------------------------------------------}
procedure UnBlock_Blocked_Siblings;

var
  Finished       : Boolean;
  First_Sibling,
  Second_Sibling : List_Ptr;
  Previous_Node,
  Temp_Node      : PQ_NOde_Ptr;

begin
  Temp_Node := NBour_of (Current_Node);
  if Temp_Node^. Mark <> Blocked          { get the Sibling that is Blocked }
      then Temp_Node := Other_NBour_of (Current_Node);
  Previous_Node := Current_Node;
  Finished := false;
  while not Finished do       { walk through the Block, UnBlocking them }
    begin
      Temp_Node^. Mark := UnBlocked;
      Remove_from_Blocked_List (Temp_Node);
      Blocked_Nodes := Blocked_Nodes - 1;
      Temp_Node^. Parent := Current_Parent;        { Unblock the node }
      Current_Parent^. Pert_Child_Count :=
            Current_Parent^. Pert_Child_Count + 1; { increment Pertinent Child Count }
      Walk_Normal (Previous_Node, Temp_Node);
      if Temp_Node = Nil
        then Finished := true;
      if not Finished
        then Finished := not (Temp_Node^. Mark = Blocked)   { check if we reached end of block }
    end
end;
```

```
{-------------------------------------------------------}
{- Main Bubble Procedure.                               -}
{-------------------------------------------------------}
begin
  Initialise_Bubble;          { Initialise }
  Place_Leaves_On_Queue;      { Start with the Leaves }
  if Sizeof_Queue = 1
    then Remove_from_Queue (Current_Node);  { This is already valid }
  while (Sizeof_Queue + Block_Count + Off_the_Top) > 1 do
    begin
      if Sizeof_Queue = 0
        then begin
          Planar := false;
          Exit
        end;
      Remove_from_Queue (Current_Node);       { Get the current node off the Queue }
      Block_or_Unblock_Node;                  { Determine marking for Node.        }
      if Current_Node^. Mark = UnBlocked
        then begin
          Current_Parent := Current_Node^. Parent;   { get the parent }
          if Blocked_Siblings > 0
            then UnBlock_Blocked_Siblings;           { Unblock siblings if necessary }
          if Current_Parent = Nil
            then Off_the_Top := 1                    { Note we have reached top of Tree }
            else begin
              Current_Parent^. Pert_Child_Count :=
                  Current_Parent^. Pert_Child_Count + 1;
              if Current_Parent^. Mark = None        { if we have not Queued the Parent, then do so }
                then begin
                  Add_to_Queue (Current_Parent);
                  Current_Parent^. Mark := Queued
                end
            end;
          Block_Count := Block_Count - Blocked_Siblings;  { decrement the number of blocks }
        end
        else begin                                          { A new Blocked Sibling }
          Block_Count := Block_Count + 1 - Blocked_Siblings; { Increase Block_Count appropriately }
          Blocked_Nodes := Blocked_Nodes + 1;
          Add_to_Blocked_List (Current_Node)                { and add the node to Blocked list }
        end
    end;
  if (Block_Count = 1) and (Blocked_Nodes > 1)   { Pseudo Node case }
    then Create_Pseudo_Node
    else begin
      if Block_Count > 1
        then Planar := false             { more than one block of blocked siblings }
        else if Blocked_Nodes = 1
               then dispose (Blocked_List);
      Pseudo_Node := Nil
    end
end;

end.
```

```
{-------------------------------------------------------------}
{- The Reduction Unit of Lempel, Even and Cederbaum.         -}
{- In this unit the Reduction phase is carried out.          -}
{-------------------------------------------------------------}
unit Lempel_Reduce;

interface

procedure Reduce_Pertinent_Subtree;   { single procedure to perform the reduction }

implementation

uses
  Planar_Defs,
  Lempel_Globals;

{-------------------------------------------------------------}
{- We perform the reduction by matching Templates.           -}
{-------------------------------------------------------------}
procedure Reduce_Pertinent_Subtree;

var
  Sizeof_Set_S    : Integer;      { The size of the pertinent leaf set }
  Current_Parent,
  Current_Node    : PQ_Node_Ptr;

{-------------------------------------------------------------}
{- Returns the number of partial kids a particular node has.-}
{-------------------------------------------------------------}
function Count_Partial_Kids (Node : PQ_Node_Ptr) : Integer;

var
  List_Element : List_Ptr;
  Count        : Integer;

begin
  List_Element := Current_Node^. Partial_Kids;
  Count := 0;
  while (List_Element <> Nil) do     { for every partial node do }
    begin
      Count := Count + 1;                  { increment the count }
      List_Element := List_Element^. Next   { and go to next partial node }
    end;
  Count_Partial_Kids := Count        { return the total }
end;
```

```
{-------------------------------------------------------}
{- A single Full child at List_Posn in the full list is     -}
{- removed from Parent_Node's kids and added to Full_Node's.-}
{-------------------------------------------------------}
procedure Remove_Add (var List_Posn    : List_Ptr;
                      var Parent_Node,
                          Full_Node    : PQ_Node_Ptr);

var Chain_Posn : Double_Ptr;

begin
  Chain_Posn := List_Posn^. Element^. Circ_List_Posn;
  if Chain_Posn^. Right = Chain_Posn    { special case, the only kid }
    then Parent_Node^. List_Start := Nil
    else begin                          { otherwise delete the node as usual }
      Chain_Posn^. Right^. Left := Chain_Posn^. Left;
      Chain_Posn^. Left^. Right := Chain_Posn^. Right;
      if Parent_Node^. List_Start = Chain_Posn
        then Parent_Node^. List_Start := Chain_Posn^. Right;
    end;
  Add_to_Circle_Link (Full_Node, Chain_Posn^. Element);
  dispose (Chain_Posn)
end;

{-------------------------------------------------------}
{- The full nodes are stripped from From_Node and added as  -}
{- kids of Full_Node returned.                              -}
{-------------------------------------------------------}
procedure Strip_Full_Nodes (var From_Node, Full_Node : PQ_Node_Ptr);

var
  Temp_Double  : Double_Ptr;
  Temp_List    : List_Ptr;

begin
  if From_Node^. Full_Kids_Count > 1              { almost no work if only 1 full kid }
    then begin
      Create_PQ_Node (Full_Node);
      with Full_Node^ do
        begin
          Data_Label := Full;       { new P-node }
          Node_Type := P_Node;
          Child_Count := From_Node^. Full_Kids_Count;    { set correct kid count }
          List_Start := Nil;
          Temp_List := From_Node^. Full_Kids;        { for every full kid do }
          while Temp_List <> Nil do    { and remove from From_Node and add to new P-node }
            begin
              Remove_Add (Temp_List, From_Node, Full_Node);
              Temp_List := Temp_List^. Next
            end;
          Full_Kids := From_Node^. Full_Kids;
          Full_Kids_Count := From_Node^. Full_Kids_Count;
        end
    end
    else begin                                    { only one full kid case }
      Full_Node := From_Node^. Full_Kids^. Element;  { it becomes the new 'P-node' }
      Temp_Double := Full_Node^. Circ_List_Posn;
      if Temp_Double^. Right = Temp_Double          { remove it from the list }
```

```
        then From_Node^. List_Start := nil
        else begin
          Temp_Double^. Right^. Left := Temp_Double^. Left;
          Temp_Double^. Left^. Right := Temp_Double^. Right;
          if From_Node^. List_Start = Temp_Double
            then From_Node^. List_Start := Temp_Double^. Right
        end;
      dispose (Temp_Double)
    end;
  From_Node^. Child_Count := From_Node^. Child_Count -   { adjust child count accordingly }
                                    From_Node^. Full_Kids_Count;
  From_Node^. Full_Kids_Count := 0;      { Note no full kids }
  From_Node^. Full_Kids := Nil;          { and list is empty }
end;

{-------------------------------------------------------------}
{- Node1 and Node2 are made siblings.                         -}
{-------------------------------------------------------------}
procedure Add_Sibling (Node1, Node2 : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Next := Node1^. Immediate_Siblings;
  List_Element^. Element := Node2;
  Node1^. Immediate_Siblings := List_Element;
  new (List_Element);
  List_Element^. Next := Node2^. Immediate_Siblings;
  List_Element^. Element := Node1;
  Node2^. Immediate_Siblings := List_Element
end;

{-------------------------------------------------------------}
{- New1 had Old1 as a Sibling, now has New2.                  -}
{- New2 had Old2 as a Sibling, now has New1.                  -}
{-------------------------------------------------------------}
procedure Replace_Replace_Siblings (New1, Old1, New2, Old2 : PQ_Node_Ptr);

begin
  with New1^. Immediate_Siblings^ do
    if Element = Old1
      then Element := New2
      else Next^. Element := New2;
  with New2^. Immediate_Siblings^ do
    if Element = Old2
      then Element := New1
      else Next^. Element := New1
end;
```

```
{-------------------------------------------------------------}
{- Counts number of times a full kid that has no full       -}
{- sibling or no sibling at all (i.e. endmost).             -}
{-------------------------------------------------------------}
procedure Count_end_Full_Kids (var Parent_Node : PQ_Node_Ptr; var Count : integer);

var
  Temp_Kid     : PQ_Node_Ptr;
  List_Element : List_Ptr;

begin
  Count := 0;
  List_Element := Parent_Node^. Full_Kids;
  while (Count < 3) and (List_Element <> Nil) do    { anything more than 2 is illegal }
    begin
      Temp_Kid := List_Element^. Element;
      if NBour_of (Temp_Kid)^. Data_Label <> full   { check neighbours }
            then Count := Count + 1;
      Temp_Kid := Other_NBour_of (Temp_Kid);
      if (Temp_Kid = Nil) or (Temp_Kid^. Data_Label <> Full)
            then Count := Count + 1;
      List_Element := List_Element^. Next
    end;
end;


{-------------------------------------------------------------}
{- The Full and Empty endmost Kids of a Q node are found.   -}
{-------------------------------------------------------------}
procedure Get_Full_Empty_Children (var Node,
                                   Full_Child, Empty_Child : PQ_Node_Ptr);

begin
  if Node <> Nil
    then begin
      if Node^. RightMost_Kid^. Data_Label <> Empty
        then begin
          Full_Child := Node^. RightMost_Kid;
          Empty_Child := Node^. LeftMost_Kid;
        end
        else begin
          Full_Child := Node^. LeftMost_Kid;
          Empty_Child := Node^. RightMost_Kid;
        end
    end
    else begin
      Full_Child := Nil;
      Empty_Child := Nil
    end
end;
```

```
{-------------------------------------------------------}
{- To follow we code all the Templates.                -}
{-------------------------------------------------------}

{-------------------------------------------------------}
{- Template L1 - A single Leaf.                        -}
{-------------------------------------------------------}
function Template_L1 (Current_Node : PQ_Node_Ptr) : Boolean;

var List_Element : List_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> Leaf
      then Template_L1 := false
      else begin
        Data_Label := full;   { Note that the Leaf is Full }
        if Root_Node = Nil    { If we are not at the Pertinent Root }
          then begin
            Parent^. Full_Kids_Count := Parent^. Full_Kids_Count + 1; { Note an extra kid }
            new (List_Element);
            List_Element^. Element := Current_Node;
            List_Element^. Next := Parent^. Full_Kids;
            Parent^. Full_Kids := List_Element
          end;
        Template_L1 := true
      end;
end;

{-------------------------------------------------------}
{- Template P1 - A P Node whose kids are all Full.     -}
{-------------------------------------------------------}
function Template_P1 (Current_Node : PQ_Node_Ptr) : Boolean;

var List_Element : List_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> P_Node
      then Template_P1 := false
      else if Full_Kids_Count <> Child_Count   { Check if kids are all full }
             then Template_P1 := false
             else begin
               Data_Label := full;       { Note it is also full }
               if Root_Node = Nil        { Check if we are at the Pertinent Root }
                 then begin
                   Parent^. Full_Kids_Count :=    { If not we add extra full kid to parent }
                     Parent^. Full_Kids_Count + 1;
                   new (List_Element);
                   List_Element^. Element := Current_Node;
                   List_Element^. Next := Parent^. Full_Kids;
                   Parent^. Full_Kids := List_Element;
                   Add_List_to_Used_List (Current_Node^. Full_Kids);  { Add Full Kids List to Used list }
                   Current_Node^. Full_Kids := Nil;
                 end;
               Template_P1 := true
             end
end;
```

```
{-------------------------------------------------------}
{- Template P2 - Must be at the Pertinent Root - A P-Node   -}
{- some of the Kids are full, the rest are empty.           -}
{-------------------------------------------------------}
function Template_P2 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Full_Node  : PQ_Node_Ptr;

begin
  With Current_Node^ do
    if (Node_Type <> P_Node) or (Partial_Kids <> Nil)
      then Template_P2 := false
      else begin
        if Full_Kids_Count = 1            { Return that node }
          then Root_Node := Full_Kids^. Element
          else begin
            Strip_Full_Nodes (Current_Node, Full_Node);       { Delete all the full nodes from Current_Node
            Add_to_Circle_Link (Current_Node, Full_Node);     { Re-Add Full node to Current_Node }
            Current_Node^. Child_Count := Current_Node^. Child_Count + 1;
            Full_Node^. Parent := Current_Node;
            Root_Node := Full_Node;                           { and return the new Pertinent Root }
          end;
        Add_Node_to_Used_List (Current_Node);
        Template_P2 := true
      end
end;

{-------------------------------------------------------}
{- Template P3 - Must NOT be at the Pertinent Root          -}
{- A P-Node with some Kids are full, the rest are empty.    -}
{-------------------------------------------------------}
function Template_P3 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element : List_Ptr;
  Empty_Node,
  Full_Node,
  New_Root     : PQ_Node_Ptr;
  Number_Empty : Integer;

begin
  with Current_Node^ do
    if (Node_Type <> P_Node) or (Partial_Kids <> Nil)
      then Template_P3 := false
      else begin
        Create_PQ_Node (New_Root);            { The new Parent is a Q-Node }
        New_Root^. Node_Type := Q_Node;
        Number_Empty := Child_Count - Full_Kids_Count;
        Replace_Node_Partial (Current_Node, New_Root, false);   { Replace Current_Node by New_Root }
                                                                { in Parent and Sibling chains     }
        Strip_Full_Nodes (Current_Node, Full_Node);     { Now strip all full kids from Current_Node }
        new (List_Element);
        List_Element^. Element := Full_Node;
        List_Element^. Next := New_Root^. Full_Kids;    { Add Full_Node to Full_Kids of New_Root }
        New_Root^. Full_Kids := List_Element;
        New_Root^. Full_Kids_Count := 1;                { and note count of 1 }
```

```
        Full_Node^. Parent := New_Root;                 { Add Full_Node to kids }
        if Number_Empty = 1                             { Note which Node is the empty node }
          then Empty_Node := List_Start^. Element       { if there is only 1 empty, then we want }
                                                        { to avoid chains - so delete Current_Node }
          else begin
            Empty_Node := Current_Node;
            Empty_Node^. Child_Count := Number_Empty;
            Empty_Node^. Data_Label := Empty
          end;
        Empty_Node^. Parent := New_Root;                { Empty_Node's parent is New_Root }
        Add_Sibling (Empty_Node, Full_Node);            { They are Siblings }
        Full_Node^. Data_Label := Full;
        New_Root^. RightMost_Kid := Full_Node;          { They are also endmost kids }
        New_Root^. LeftMost_Kid := Empty_Node;
        if Number_Empty < 2                             { we want to avoid chains }
          then begin
            dispose (Current_Node);
            Current_Node := Nil
          end
          else Add_Node_to_Used_List (Current_Node);
        Add_List_to_Used_List (Full_Node^. Full_Kids);
        Full_Node^. Full_Kids := Nil;
        Template_P3 := true
      end;
end;

{-------------------------------------------------------------}
{- Template P4 - Must be at the Pertinent Root, A P-Node    -}
{- with one Partial Q-Node and possibly some Kids are full, -}
{- possibly some are empty.                                 -}
{-------------------------------------------------------------}
function Template_P4 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element : List_Ptr;
  Full_Node,
  Only_Partial : PQ_Node_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> P_Node
      then Template_P4 := false
      else
        if Count_Partial_Kids (Current_Node) <> 1    { Must have exactly 1 partial kid }
          then Template_P4 := false
          else begin
            Only_Partial := Partial_Kids^. Element;  { This is our new Root }
            if (Only_Partial^. LeftMost_Kid^. Data_Label <> Full)
                  and (Only_Partial^. RightMost_Kid^. Data_Label <> Full)
              then Template_P4 := false       { Must have a full element at endmost }
              else begin
                if Full_Kids_Count > 0        { Get rid of full kids on current }
                  then begin
                    Strip_Full_Nodes (Current_Node, Full_Node);      { new parent P-node }
                    Add_List_to_Used_List (Full_Node^. Full_Kids);
                    Full_Node^. Full_Kids := Nil;
                    Full_Node^. Parent := Only_Partial;              { add new to partial's kids }
                    new (List_Element);
```

```
                    List_Element^. Next := Only_Partial^. Full_Kids;
                    List_Element^. Element := Full_Node;
                    Only_Partial^. Full_Kids := List_Element;
                    Only_Partial^. Full_Kids_Count := Only_Partial^. Full_Kids_Count + 1;
                    if Only_Partial^. Leftmost_Kid^. Data_Label = Full    { and adjust endmost kids }
                      then begin
                        Add_Sibling (Only_Partial^. Leftmost_Kid, Full_Node);
                        Only_Partial^. Leftmost_Kid := Full_Node
                      end
                      else begin
                        Add_Sibling (Only_Partial^. Rightmost_Kid, Full_Node);
                        Only_Partial^. Rightmost_Kid := Full_Node
                      end
                  end;
                Root_Node := Only_Partial;    { Note the new pertinent Root }
                dispose (Current_Node^. Partial_Kids);
                Template_P4 := true
              end
          end
end;

{-------------------------------------------------------}
{- Template P5 - Must NOT be at the Pertinent Root, A P-Node-}
{- with one Partial Q-Node and possibly some Kids are full, -}
{- possibly some are empty.                                -}
{-------------------------------------------------------}
function Template_P5 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element        : List_Ptr;
  Temp_Double         : Double_Ptr;
  Number_Empty        : Integer;
  New_Root,
  Full_Node,
  Empty_Node,
  Empty_EndMost_Child,
  Full_EndMost_Child  : PQ_Node_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> P_Node
      then Template_P5 := false
      else
        if Count_Partial_Kids (Current_Node) <> 1 { must have exactly 1 Partial Kid }
          then Template_P5 := false
          else begin
            New_Root := Partial_Kids^. Element;    { this node becomes the Root of the replacement }
            Number_Empty := Current_Node^. Child_Count -
                            Current_Node^. Full_Kids_Count -
                              Count_Partial_Kids (Current_Node);
            Get_Full_Empty_Children (New_Root,     { get the endmost kids }
                                     Full_Endmost_Child, Empty_Endmost_Child);
            Replace_Node_Partial (Current_Node, New_Root, true);   { replace Current_Node with New_Root }
            if Full_Kids_Count > 0                                 { get rid of full kids of Current_Node }
              then begin
                Strip_Full_Nodes (Current_Node, Full_Node);   { get rid of full kids from Current }
                Add_List_to_Used_List (Full_Node^. Full_Kids);
                Full_Node^. Full_Kids := Nil;
```

```
                Full_Node^. Parent := New_Root;
                new (List_Element);
                List_Element^. Next := New_Root^. Full_Kids;  { add them to new partial's kids }
                List_Element^. Element := Full_Node;
                New_Root^. Full_Kids := List_Element;
                New_Root^. Full_Kids_Count := New_Root^. Full_Kids_Count + 1;
                if New_Root^. Leftmost_Kid^. Data_Label = Full { and adjust endmost kids }
                  then begin
                    Add_Sibling (New_Root^. Leftmost_Kid, Full_Node);
                    New_Root^. Leftmost_Kid := Full_Node
                  end
                  else begin
                    Add_Sibling (New_Root^. Rightmost_Kid, Full_Node);
                    New_Root^. Rightmost_Kid := Full_Node
                  end;
              end;
            if Number_Empty > 0          { Note the empty node }
              then begin
                if Number_Empty = 1      { As usual, watch for chains }
                  then begin
                    Temp_Double := Current_Node^. List_Start;
                    while Temp_Double^. Element^. Data_Label <> Empty do  { get first empty node }
                      Temp_Double := Temp_Double^. Right;
                    Empty_Node := Temp_Double^. Element
                  end
                  else begin                  { more than 1 empty - so current_node will do }
                    Empty_Node := Current_Node;
                    Empty_Node^. Data_Label := Empty;
                    Empty_Node^. Child_Count := Number_Empty
                  end;
                Empty_Node^. Parent := New_Root;               { New_Root is its new parent }
                Add_Sibling (Empty_EndMost_Child, Empty_Node);  { Add Sibling pointers }
                if New_Root^. LeftMost_Kid^. Data_Label = Empty { Adjust endmost kids }
                  then New_Root^. LeftMost_Kid := Empty_Node
                  else New_Root^. RightMost_Kid := Empty_Node;
              end;
            dispose (Partial_Kids);
            if Number_Empty < 2          { cleanup for chains }
              then begin
                dispose (Current_Node);
                Current_Node := Nil
              end
              else Add_Node_to_Used_List (Current_Node);
            Template_P5 := true
          end
end;
```

```
{-------------------------------------------------------}
{- Template P6 - Must be at the Pertinent Root, A P-Node    -}
{- with two Partial Q-Nodes and possibly some Kids are full,-}
{- possibly some are empty.                                 -}
{-------------------------------------------------------}
function Template_P6 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Number_Empty      : Integer;
  Temp_List         : List_Ptr;
  Temp_Double       : Double_Ptr;
  Partial1,
  Partial2,
  Full_Node,
  Full_Kid_Partial1,
  Empty_Kid_Partial2,
  Full_Kid_Partial2 : PQ_Node_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> P_Node
      then Template_P6 := false
      else
        if Count_Partial_Kids (Current_Node) <> 2
          then Template_P6 := false
          else begin
            Partial1 := Partial_Kids^. Element;          { Partial1 becomes the New_Root }
            Partial2 := Partial_Kids^. Next^. Element;
            Number_Empty := Child_Count - Full_Kids_Count - 2;
            if Full_Kids_Count > 0                        { strip Full_Nodes off Current_Node }
              then begin
                Strip_Full_Nodes (Current_Node, Full_Node);
                Add_List_to_Used_List (Full_Node^. Full_Kids);
                new (Temp_List);
                Temp_List^. Element := Full_Node;
                Temp_List^. Next := Partial1^. Full_Kids;
                Partial1^. Full_Kids := Temp_List;
                Partial1^. Full_Kids_Count := Partial1^. Full_Kids_Count + 1;
              end
              else Full_Node := Nil;
            Get_Full_Empty_Children (Partial2,           { get Partial2's endmost kids }
                                     Full_Kid_Partial2, Empty_Kid_Partial2);
            Empty_Kid_Partial2^. Parent := Partial1;      { will become Partial1's endmost empty }
            if Partial1^. LeftMost_Kid^. Data_Label = Full     { adjust endmost kid pointers }
              then begin
                Full_Kid_Partial1 := Partial1^. LeftMost_Kid;
                Partial1^. LeftMost_Kid := Empty_Kid_Partial2
              end
              else begin
                Full_Kid_Partial1 := Partial1^. RightMost_Kid;
                Partial1^. RightMost_Kid := Empty_Kid_Partial2
              end;
            if Full_Node = Nil
              then Add_Sibling (Full_Kid_Partial1, Full_Kid_Partial2)
              else begin
                Add_Sibling (Full_Kid_Partial1, Full_Node);
                Add_Sibling (Full_Kid_Partial2, Full_Node)
              end;
```

```
            Temp_Double := Partial2^. Circ_List_Posn;          { delete partial2 from circ link }
            Temp_Double^. Right^. Left := Temp_Double^. Left;
            Temp_Double^. Left^. Right := Temp_Double^. Right;
            Child_Count := Child_Count - 1;                    { Count declines accordingly }
            if List_Start = Temp_Double
              then List_Start := Temp_Double^. Right;
            dispose (Temp_Double);                             { Claim memory now unused }
            Temp_List := Partial2^. Full_Kids;
            while Temp_List^. Next <> Nil do
              Temp_List := Temp_List^. Next;
            Temp_List^. Next := Partial1^. Full_Kids;
            Partial1^. Full_Kids := Partial2^. Full_Kids;
            Partial1^. Full_Kids_Count := Partial1^. Full_Kids_Count + Partial2^. Full_Kids_Count;
            dispose (Partial2);                                { Claim memory now unused }
            dispose (Partial_Kids^. Next);
            dispose (Partial_Kids);
            Root_Node := Partial1;
            if Number_Empty = 0                                { Check for chains }
              then begin
                Replace_Node_Partial (Current_Node, Root_Node, false);   { Replace with Root_Node }
                dispose (Current_Node);
                Current_Node := Nil
              end
              else Add_Node_to_Used_List (Current_Node);
            Template_P6 := true
          end;
end;

{-------------------------------------------------------------}
{- Template Q1 - A Q-Node whose kids are all full.          -}
{-------------------------------------------------------------}
function Template_Q1 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Temp_List : List_Ptr;
  Count     : integer;

begin
  with Current_Node^ do
    if (Node_Type <> Q_Node) or (Current_Node = Pseudo_Node)    { Pseudo Node is always Q3 }
      then Template_Q1 := false
      else begin
        if (RightMost_Kid^. Data_Label <> Full)
          or (LeftMost_Kid^. Data_Label <> Full)
          then Template_Q1 := false
          else begin
            Count_end_Full_Kids (Current_Node, Count);
            if (Count <> 2) or (Rightmost_Kid^. Data_Label <> Full)
                  or (Leftmost_Kid^. Data_Label <> Full)
              then Template_Q1 := false
              else begin
                Data_Label := full;                    { all kids are full, so Current_Node is full }
                if Root_Node = Nil
                  then begin                           { update Parent's full kids data }
                    Parent^. Full_Kids_Count :=
                      Parent^. Full_Kids_Count + 1;
                    new (Temp_List);
                    Temp_List^. Element := Current_Node;    { add current to full kids }
```

```
                Temp_List^. Next := Parent^. Full_Kids; { of the parent          }
                Parent^. Full_Kids := Temp_List;
              end;
            Template_Q1 := true
          end
        end
    end
end;

{-------------------------------------------------------}
{- Template Q2 - A Q-Node some of whose kids are full,      -}
{-                       at most one partial kid,          -}
{-                       and some kids are empty.          -}
{- If there are any Full kids, they must all be at one end. -}
{- The partial kid MUST follow the full kids.              -}
{-------------------------------------------------------}
function Template_Q2 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Count         : Integer;
  End_Kid,
  Partial_Child,
  Full_Child,
  Empty_Child,
  Full_Sibling,
  Empty_Sibling : PQ_Node_Ptr;
  Temp_List     : List_Ptr;

begin
  with Current_Node^ do
    if (Node_Type <> Q_Node) or (Current_Node = Pseudo_Node)    { Pseudo Node is Q3 }
      then Template_Q2 := false
      else
        if Count_Partial_Kids (Current_Node) > 1     { at most 1 partial kid }
          then Template_Q2 := false
          else begin
            if Partial_Kids <> Nil
              then Partial_Child := Partial_Kids^. Element   { get the full kid }
              else Partial_Child := Nil;
            if Full_Kids_Count > 0
              then begin
                Count_end_Full_Kids (Current_Node, Count);
                if RightMost_Kid^. Data_Label = Full       { get the endmost kid }
                  then End_Kid := RightMost_Kid
                  else
                    if LeftMost_Kid^. Data_Label = Full
                      then End_Kid := LeftMost_Kid
                      else End_Kid := Nil;
                if (End_Kid = Nil)
                        or (Count <> 2)
                  then begin
                    Template_Q2 := false;
                    exit
                  end;
                if Partial_Child <> Nil
                  then begin
                    Get_Full_Empty_Siblings (Partial_Child,
                                             Full_Sibling, Empty_Sibling);
```

```
            if (Full_Sibling^. Data_Label <> Full) { if there is a partial kid, }
              then begin                           { then it must be neighbour  }
                Template_Q2 := false;
                Exit
              end;
          end
        end
        else      { there are no Full Kids }
          if (Partial_Child <> RightMost_Kid)    { Check if the Partial kid is endmost }
            and (Partial_Child <> LeftMost_Kid)
            then begin
              Template_Q2 := false;
              Exit
            end
            else Get_Full_Empty_Siblings (Partial_Child,
                                          Full_Sibling, Empty_Sibling);

      Data_Label := Partial;    { Everything is now checked, Template Q2 is matched }
      new (Temp_List);
      Temp_List^. Next := Parent^. Partial_Kids;   { Add to Partial list of parent }
      Temp_List^. Element := Current_Node;
      Parent^. Partial_Kids := Temp_List;
      if Partial_Child <> Nil    { must join sibling links between Partial Child's }
        then begin               { kids and Partial_Child's Siblings               }
          Get_Full_Empty_Children (Partial_Child,
                                   Full_Child, Empty_Child);   { Get endmost kids }
          if Full_Sibling <> Nil
            then
              Add_Replace_Sibling (Full_Child, Partial_Child,
                                   Full_Sibling)
            else begin
              if RightMost_Kid = Partial_Child     { adjust endmost pointers   }
                then RightMost_Kid := Full_Child   { to be Partial_Child's kid }
                else LeftMost_Kid := Full_Child;
              Full_Child^. Parent := Current_Node  { set proper parent pointer }
            end;
          if Empty_Sibling <> Nil                  { set up link to Partial Child's kid }
            then Add_Replace_Sibling (Empty_Child, Partial_Child,
                                      Empty_Sibling)
            else begin                             { if no empty Sibling then }
              if RightMost_Kid = Partial_Child     { there are only full kids }
                then RightMost_Kid := Empty_Child  { so Partial_Child was endmost }
                else LeftMost_Kid := Empty_Child;
              Empty_Child^. Parent := Current_Node { set new endmost Parent pointer }
            end;
          if Full_Kids <> Nil
            then begin
              Temp_List := Partial_Child^. Full_Kids;
              while Temp_List^. Next <> Nil do
                Temp_List := Temp_List^. Next;
              Temp_List^. Next := Full_Kids;
              Full_Kids := Partial_Child^. Full_Kids;
              Full_Kids_Count := Full_Kids_Count + Partial_Child^. Full_Kids_Count
            end
            else begin
              Full_Kids := Partial_Child^. Full_Kids;
              Full_Kids_Count := Partial_Child^. Full_Kids_Count
            end;
```

```
              Kill_Node (Partial_Child)
            end;
          Template_Q2 := true
        end
end;

{-------------------------------------------------------}
{- Template Q3 - A Q-Node some of whose kids are full,  -}
{-                  at most two partial kids,          -}
{-                  and some kids are empty.           -}
{- If there are any Full kids, they must all be between the -}
{- two partial kids, or next to the one.               -}
{-------------------------------------------------------}
function Template_Q3 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Count                          : Integer;
  Partial_Child_1, Partial_Child_2,
  Empty_Sibling1, Full_Sibling1,
  Empty_Sibling2, Full_Sibling2,
  Full_Child, Empty_Child        : PQ_Node_Ptr;
  Temp_List                      : List_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> Q_Node
      then Template_Q3 := false
      else
        if Count_Partial_Kids (Current_Node) > 2   { at most 2 partial kids }
          then Template_Q3 := false
          else begin
            if Partial_Kids <> Nil        { set up the partial kids }
              then begin
                Partial_Child_1 := Partial_Kids^. Element;
                if Partial_Kids^. Next <> Nil
                  then Partial_Child_2 := Partial_Kids^. Next^. Element
                  else Partial_Child_2 := Nil
              end
              else Partial_Child_1 := Nil;
            if (Full_Kids_Count > 0)      { now start to match }
              then begin
                Count_end_Full_Kids (Current_Node, Count);
                if (Count > 2)            { check for separated full kids }
                  then begin
                    Template_Q3 := false;
                    Exit
                  end;
                if Partial_Child_1 <> Nil { check partials are adjacent }
                  then begin
                    Get_Full_Empty_Siblings (Partial_Child_1,
                                             Full_Sibling1, Empty_Sibling1);
                    if Full_Sibling1^. Data_Label <> Full
                      then begin
                        Template_Q3 := false;
                        exit
                      end;
                    if Partial_Child_2 <> Nil    { both must be adjacent if there are two }
                      then begin
```

```
                Get_Full_Empty_Siblings (Partial_Child_2,
                                         Full_Sibling2, Empty_Sibling2);
                if Full_Sibling2^. Data_Label <> Full
                  then begin
                    Template_Q3 := false;
                    exit
                  end;
              end
          end
        end
        else                              { No full kids - two partials }
          if Count_Partial_Kids (Current_Node) = 2   { then they must be adjacent }
            then begin
              Get_Full_Empty_Siblings (Partial_Child_1,
                                       Full_Sibling1, Empty_Sibling1);
              if Full_Sibling1 <> Partial_Child_2
                then begin
                  Template_Q3 := false;
                  Exit
                end
            end;
      if Partial_Kids <> Nil       { nothing to do if no partial kids }
        then begin
          Data_Label := Partial;   { Template match is Confirmed }
          Get_Full_Empty_Children (Partial_Child_1,
                                   Full_Child, Empty_Child);
          if Empty_Sibling1 <> Nil      { make the links }
            then Add_Replace_Sibling (Empty_Child, Partial_Child_1,
                                      Empty_Sibling1)
            else Adjust_End_Most_Kids (Current_Node, Partial_Child_1,
                                       Empty_Child);
          if Full_Sibling1 <> Nil      { make the links }
            then Add_Replace_Sibling (Full_Child, Partial_Child_1,
                                      Full_Sibling1)
            else Adjust_End_Most_Kids (Current_Node, Partial_Child_1, Full_Child);
          if Partial_Child_2 <> Nil      { do the same for Partial2 }
            then begin
              Get_Full_Empty_Siblings (Partial_Child_2,
                                       Full_Sibling2,
                                       Empty_Sibling2);
              Get_Full_Empty_Children (Partial_Child_2,
                                       Full_Child, Empty_Child);
              if Empty_Sibling2 <> Nil
                then Add_Replace_Sibling (Empty_Child, Partial_Child_2,
                                          Empty_Sibling2)
                else Adjust_End_Most_Kids (Current_Node, Partial_Child_2, Empty_Child);
              if Full_Sibling2 <> Nil      { make the links }
                then Add_Replace_Sibling (Full_Child, Partial_Child_2,
                                          Full_Sibling2)
                else Adjust_End_Most_Kids (Current_Node, Partial_Child_2, Full_Child);
            end;
          if Full_Kids = Nil                    { and update full kids list }
            then begin
              Full_Kids := Partial_Child_1^. Full_Kids;
              Full_Kids_Count := Partial_Child_1^. Full_Kids_Count
            end
            else begin
              Temp_List := Partial_Child_1^. Full_Kids;
```

```
                    while Temp_List^. Next <> nil do
                      Temp_List := Temp_List^. Next;
                    Temp_List^. Next := Full_Kids;
                    Full_Kids := Partial_Child_1^. Full_Kids;
                    Full_Kids_Count := Full_Kids_Count + Partial_Child_1^. Full_Kids_Count
                  end;
                if Partial_Child_2 <> Nil          { update full kids list }
                  then begin
                    Temp_List := Partial_Child_2^. Full_Kids;
                    while Temp_List^. Next <> nil do
                      Temp_List := Temp_List^. Next;
                    Temp_List^. Next := Full_Kids;
                    Full_Kids := Partial_Child_2^. Full_Kids;
                    Full_Kids_Count := Full_Kids_Count + Partial_Child_2^. Full_Kids_Count
                  end;
                Kill_Node (Partial_Child_1);    { cleanup memory allocations }
                if Partial_Child_2 <> Nil
                  then begin
                    Kill_Node (Partial_Child_2);
                    dispose (Partial_Kids^. Next)
                  end;
                dispose (Partial_Kids);
              end;
            Template_Q3 := true
          end;
end;

{-------------------------------------------------------}
{- Main Initialise Routine                             -}
{-------------------------------------------------------}
procedure Initialise_Reduction;

var
  Dummy_PQ     : PQ_Node_Ptr;

begin
  if Queue_Start <> Nil
    then repeat
      Remove_from_Queue (Dummy_PQ)     { Cleanup after Bubble Phase }
    until Queue_Start = Nil;
  Root_Node := Nil
end;
```

```
begin
  Initialise_Reduction;                          { Do main Init }
  Place_Leaves_On_Queue;                         { Place pertinent leaves on Queue }
  Sizeof_Set_S := Sizeof_Queue;
  while (Sizeof_Queue > 0) and (Planar) do
    begin
      Remove_from_Queue (Current_Node);
      if Current_Node^. Node_Type = Leaf
        then Current_Node^. Pert_Leaf_Count := 1;
      if Current_Node^. Pert_Leaf_Count < Sizeof_Set_S    { Check if we are at pertinent Root }
        then begin                          { if not, then }
          Current_Parent := Current_Node^. Parent;
          Current_Parent^. Pert_Leaf_Count :=     { update Parent's Pert_Leaf_Count of Pertinent Leaves }
            Current_Parent^. Pert_Leaf_Count + Current_Node^. Pert_Leaf_Count;
          Current_Parent^. Pert_Child_Count :=    { Update number of kids left to process }
            Current_Parent^. Pert_Child_Count - 1;
          if Current_Parent^. Pert_Child_Count = 0  { see if all kids matched }
            then Add_to_Queue (Current_Parent);
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P3 (Current_Node) then
          if not Template_P5 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
            begin
              writeln ('Halting no template matches');
              Planar := false
            end
        end
        else begin       { This node is the Pertinent Root }
          Root_Node := Current_Node;              { default is this node is Pertinent Root }
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P2 (Current_Node) then
          if not Template_P4 (Current_Node) then
          if not Template_P6 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
          if not Template_Q3 (Current_Node) then
            begin
              writeln ('Halting no template matches at pertinent root');
              Planar := false
            end
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit contains the routines use by both the Bubble   -}
{- and Reduction units of Lempel, Even and Cederbaum.       -}
{-------------------------------------------------------}
unit Lempel_Globals;

Interface

uses
  CRT,
  Planar_Miscellaneous,
  Planar_Defs;

{-------------------------------------------------------}
{- All the global variables.                            -}
{-------------------------------------------------------}
var
  ST_Number_Index    : Array [1..Max_Vertices] of Vertex_Ptr;  { used to map a St Number to }
                                                               { an actual vertex number    }
  Used_List          : List_Ptr;          { used to reset values after each reduction }
  Pseudo_Node,                            { see chapter 4 for details }
  New_Root,                               { the new root node to add new leaves to after a reduction }
  Root_Node          : PQ_Node_Ptr;       { the Root of the pertinent tree after a reduction }
  Sizeof_Queue       : Integer;           { Queue size for bubble and reduction }
  Planar             : Boolean;
  Current_Vertex,                         { Used for planarity testing of graph }
  Num_Vertices_Added : Vertex_Ptr;        { to the current component      }
  Queue_Start,                            { For the Queue                 }
  Queue_Head         : Double_Ptr;

procedure Kill_Node (var Node : PQ_Node_Ptr);               { wipe memory  allocation of Node }

procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr);  { walk between siblings }

function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;       { get an immediate sibling }

function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;  { get other immediate sibling }

procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);   { creates a new node with defaults }

procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);          { adds a node to the queue }

procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr);  { removes a node from the queue }

procedure Place_Leaves_on_Queue;                            { adds all pertinent leaves to the queue }

procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;  { adds a child to a P node parent }
                              var Child_Ptr : PQ_Node_Ptr);

procedure Add_Edges_to_Tree (From_Vertex : Vertex_Ptr;     { adds new leaves to the tree }
                             Tree_Node   : PQ_Node_Ptr);

procedure Add_Replace_Sibling (New_Node, Old_Node, Other_Node : PQ_Node_Ptr);
                        { adds Other_Node to New_Node's Sibling list and }
                        { replaces Old_Node with New_Node in Other_Node's Sibling list }

procedure Get_Full_Empty_Siblings (var Node,
                                   Full_Sibling, Empty_Sibling : PQ_Node_Ptr);
```

```
                        { Gets the full/partial sibling and empty sibling of a node }

procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);   { adds a single node to a used list }

procedure Add_List_to_Used_List (var List_Start : List_Ptr);      { adds a list of nodes to used list }

procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);
                        { Replaces in the Parent all references to Old_Node by New_Node }

Implementation

{-------------------------------------------------------}
{- Returns a PQ_Node initialised with default values    -}
{- Will not initialise fields dependent on node type.   -}
{-------------------------------------------------------}
procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);

begin
 new (PQ_Pointer);
 with PQ_Pointer^ do                { set the initial values }
   begin
     Data_Label := Empty;
     Full_Kids := Nil;
     Partial_Kids := Nil;
     Immediate_Siblings := Nil;
     Mark := None;
     Parent := Nil;
     Child_Count := 0;
     Full_Kids_Count := 0;
     Pert_Child_Count := 0;
     Pert_Leaf_Count := 0;
   end
end;

{-------------------------------------------------------}
{- Adds a PQ node to the Queue                          -}
{-------------------------------------------------------}
procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);

var Queue_Element : Double_Ptr;

begin
  new (Queue_Element);
  Queue_Element^. Left := Nil;
  Queue_Element^. Right := Queue_Start;
  Queue_Element^. Element := PQ_Element;
  if Queue_Start <> Nil
    then Queue_Start^. Left := Queue_Element;
  Queue_Start := Queue_Element;
  if Queue_Head = Nil
    then Queue_Head := Queue_Element;
  Sizeof_Queue := Sizeof_Queue + 1
end;
```

```
{-------------------------------------------------------}
{- Removes a PQ node from the Queue                    -}
{-------------------------------------------------------}
procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr);

var  Queue_Element : Double_Ptr;

begin
  if Queue_Start = Nil
    then Halt;
  Queue_Element := Queue_Head;
  Queue_Head := Queue_Head^. Left;
  if Queue_Start^. Right = Nil
    then Queue_Start := Nil;
  if Queue_Head <> Nil
    then Queue_Head^. Right := Nil;
  PQ_Element := Queue_Element^. Element;
  dispose (Queue_Element);
  Sizeof_Queue := Sizeof_Queue - 1
end;

{-------------------------------------------------------}
{- Adds all pertinent leaves to the Queue.             -}
{-------------------------------------------------------}
procedure Place_Leaves_on_Queue;

var Temp_Edge : Edge_Ptr;

begin
  Temp_Edge := Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges;  { get start edge }
  while (Temp_Edge <> Nil) do
    begin
      if (Graph^ [Temp_Edge^. Vertex]. ST_Number < Num_Vertices_Added)  { if it is part of the }
        then Add_to_Queue (Temp_Edge^. PQ_Ptr);                         { graph so far then add to queue }
      Temp_Edge := Temp_Edge^. Next
    end
end;
```

```
{-------------------------------------------------------------}
{- Adds Child_Ptr to the circular list of Parent_Node.      -}
{-------------------------------------------------------------}
procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;
                              var Child_Ptr : PQ_Node_Ptr);
var Temp_Double : Double_Ptr;

begin
  with Parent_Node^ do
    if List_Start = Nil           { check for initial case }
      then begin
        new (List_Start);
        with List_Start^ do       { set up single element circular list }
          begin
            Left := List_Start;
            Right := List_Start;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := List_Start
          end;
      end
      else begin
        new (Temp_Double);
        with Temp_Double^ do      { normal insertion into circular list }
          begin
            Left := List_Start^. Left;
            Right := List_Start;
            List_Start^. Left^. Right := Temp_Double;
            List_Start^. Left := Temp_Double;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := Temp_Double
          end
      end
end;
```

```
{-------------------------------------------------------}
{- Replace all references of Old_Node with New_Node in      -}
{- Parent. Replace it in Sibling Chains and Circular list if-}
{- necessary. Delete_Node indicates if New_Node is a child  -}
{- of Old_Node and whether it must be deleted from Old_Node -}
{- circular list.                                           -}
{- Also add New_Node to the partial children list of parent.-}
{-------------------------------------------------------}
procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);

var                              { all variables are temporary }
  Temp_Element2,
  List_Element   : List_Ptr;
  Double_Element : Double_Ptr;
  Temp_Node      : PQ_Node_Ptr;

begin
  New_Node^. Parent := Old_Node^. Parent;     { Get new parent }
  New_Node^. Pert_Leaf_Count := Old_Node^. Pert_Leaf_Count;
  New_Node^. Data_Label := Partial;        { new node is partial }
  if Root_Node = Nil                       { so add to Partial children list }
    then begin
      new (List_Element);
      List_Element^. Next := New_Node^. Parent^. Partial_Kids;
      List_Element^. Element := New_Node;
      New_Node^. Parent^. Partial_Kids := List_Element
    end;
  if Delete_Node      { We must delete Old_Node from circular list of Old_Node }
    then begin
      Double_Element := New_Node^. Circ_List_Posn;
      if Double_Element^. Left = Double_Element    { check for end condition }
        then Old_Node^. List_Start := Nil
        else begin                                 { otherwise delete as normal }
          if Double_Element = Old_Node^. List_Start
            then Old_Node^. List_Start := Double_Element^. Right;
          Double_Element^. Left^. Right := Double_Element^. Right;
          Double_Element^. Right^. Left := Double_Element^. Left;
        end;
      New_Node^. Circ_List_Posn := Nil;
      Old_Node^. Child_Count := Old_Node^. Child_Count - 1;  { note a child has been removed }
      dispose (Double_Element);
    end;
  if Old_Node^. Immediate_Siblings = Nil    { then Parent is a P Node and replace }
    then begin                              { in parent's circulra list.          }
      New_Node^. Immediate_Siblings := Nil;
      New_Node^. Circ_List_Posn := Old_Node^. Circ_List_Posn;
      New_Node^. Circ_List_Posn^. Element := New_Node;
    end
    else begin                              { Replace in immediate sibling's Siblings chain }
      New_Node^. Immediate_Siblings :=
           Old_Node^. Immediate_Siblings;.
      Old_Node^. Immediate_Siblings := Nil;
      List_Element := New_Node^. Immediate_Siblings;   { with each sibling }
      while (List_Element <> Nil) do
        begin
          Temp_Node := List_Element^. Element;
          Temp_Element2 := Temp_Node^. Immediate_Siblings;    { search fro refernce to Old_Node }
```

```
          while (Temp_Element2^. Element <> Old_Node) do
            Temp_Element2 := Temp_Element2^. Next;
          Temp_Element2^. Element := New_Node;                { replace with New_Node }
          List_Element := List_Element^. Next          { and go to next sibling }
        end;
      if New_Node^. Parent^. LeftMost_Kid = Old_Node          { adjust end most pointers as well }
        then New_Node^. Parent^. LeftMost_Kid := New_Node
        else if New_Node^. Parent^. RightMost_Kid = Old_Node
               then New_Node^. Parent^. RightMost_Kid := New_Node;
    end;
end;

{-----------------------------------------------------------}
{- Add new leaves representing edges from From_Vertex.      -}
{-----------------------------------------------------------}
procedure Add_Edges_to_Tree (From_Vertex : Vertex_Ptr;
                             Tree_Node   : PQ_Node_Ptr);

var
  New_Nodes       : Integer;            { used to check for only a single leaf }
                                        { added since we don't want chains     }
  Last_Edge,
  From_Temp_Edge  : Edge_Ptr;
  Current_ST      : Vertex_Ptr_Range;
  PQ_Element      : PQ_Node_Ptr;

begin
  New_Nodes := 0;
  From_Temp_Edge := Graph^ [From_Vertex]. Edges;
  Current_ST := Graph^ [From_Vertex]. ST_Number;
  while From_Temp_Edge <> Nil do                  { with each edge do }
    begin
      if Graph^ [From_Temp_Edge^. Vertex]. ST_Number > Current_ST   { check it goes to a higher vertex }
        then begin
          Create_PQ_Node (PQ_Element);          { create the new leaf }
          New_Nodes := New_Nodes + 1;
          with PQ_Element^ do                   { mark it as a leaf }
            begin
              Node_Type := Leaf;
              Parent := Tree_Node;
              Tail_Vertex := From_Vertex
            end;
          Tree_Node^. Child_Count := Tree_Node^. Child_Count + 1;   { add to Tree Node's kids }
          Add_to_Circle_Link (Tree_Node, PQ_Element);
          From_Temp_Edge^. PQ_Ptr := PQ_Element;              { and note which PQ Element reps this edge }
          Last_Edge := From_Temp_Edge;
          From_Temp_Edge^. Other_Edge^. PQ_Ptr := PQ_Element   { ditto for other edge element }
        end;
      From_Temp_Edge := From_Temp_Edge^. Next    { go to the next candidate edge }
    end;
  if New_Nodes = 0       { Error - ST Numbering guarantees we always can add an edge }
    then begin
      writeln ('Fatal error - no new edges to add');
      halt
    end
    else
      if New_Nodes = 1              { check for chains of a single element added }
        then with Tree_Node^ do                   { tree node becomes the leaf }
```

```
          begin
            PQ_Element := List_Start^. Element;
            dispose (List_Start);                { wipe the reference to the leaf }
            Node_Type := Leaf;                   { note this vertex is a leaf }
            Last_Edge^. PQ_Ptr := Tree_Node;       { change references to this node }
            Last_Edge^. Other_Edge^. PQ_Ptr := Tree_Node;
            Tail_Vertex := PQ_Element^. Tail_Vertex;
            dispose (PQ_Element)                 { and reclaim the node }
          end
end;

{-------------------------------------------------------------}
{- Adds Other_Node to New_Node's Sibling List.               -}
{- Also Replaces reference in Other_Node's Sibling list to   -}
{- Old_Node with New_Node.                                   -}
{-------------------------------------------------------------}
procedure Add_Replace_Sibling (New_Node, Old_Node,
                               Other_Node : PQ_Node_Ptr);
var
  List_Element : List_Ptr;

begin
  List_Element := Other_Node^. Immediate_Siblings;     { Replace phase }
  if List_Element^. Element = Old_Node
    then List_Element^. Element := New_Node
    else List_Element^. Next^. Element := New_Node;
  new (List_Element);
  List_Element^. Next := New_Node^. Immediate_Siblings;    { add reference to Other_Node }
  List_Element^. Element := Other_Node;
  New_Node^. Immediate_Siblings := List_Element
end;
```

```
{-------------------------------------------------------------}
{- The Full and Empty Siblings of a node are returned.       -}
{- For the purposes of this routine, a full node is either   -}
{- strictly full, or partial.                                -}
{-------------------------------------------------------------}
procedure Get_Full_Empty_Siblings (var Node, Full_Sibling,
                                           Empty_Sibling : PQ_Node_Ptr);

var
  Temp,
  Temp2 : PQ_Node_Ptr;

begin
  if Node <> Nil
    then begin
      Temp := Node^.Immediate_Siblings^.Element;
      if Node^. Immediate_Siblings^. Next = Nil
        then Temp2 := nil
        else Temp2 := Node^. Immediate_Siblings^. Next^. Element;
      if Temp^. Data_Label <> Empty
        then begin
          Full_Sibling := Temp;
          Empty_Sibling := Temp2
        end
        else begin
          Empty_Sibling := Temp;
          Full_Sibling := Temp2
        end
    end
    else begin
      Empty_Sibling := Nil;
      Full_Sibling := Nil
    end
end;

{-------------------------------------------------------------}
{- If one of the Endmost kids was Old_Node, then we reset    -}
{- that kid to be New_Kid.                                   -}
{-------------------------------------------------------------}
procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

begin
  with Current_Node^ do
    if RightMost_Kid = Old_Kid
      then begin
        RightMost_Kid := New_Kid;
        New_Kid^. Parent := Current_Node;
      end
      else
        if LeftMost_Kid = Old_Kid
          then begin
            LeftMost_Kid := New_Kid;
            New_Kid^. Parent := Current_Node
          end
end;
```

```
{-------------------------------------------------------}
{-  A List is added to the Used List.                  -}
{-------------------------------------------------------}
procedure Add_List_to_Used_List (var List_Start : List_Ptr);

var
  List_Element : List_Ptr;

begin
  if List_Start <> Nil
    then begin
      List_Element := List_Start;
      while List_Element^. Next <> Nil do     { go to the end of the list }
        List_Element := List_Element^. Next;
      List_Element^. Next := Used_List;       { Append the list to the front of Used list }
      Used_List := List_Start;
      List_Start := Nil
    end
end;

{-------------------------------------------------------}
{- A single node is appended to used list.             -}
{-------------------------------------------------------}
procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Next := Used_List;
  List_Element^. Element := PQ_Element;
  Used_List := List_Element
end;

{-------------------------------------------------------}
{- We walk from Current_Node to an immediate sibling that is-}
{- not Previous_Node.                                  -}
{-------------------------------------------------------}
procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr);

var
  Temp_Node : PQ_Node_Ptr;

begin
  Temp_Node := Current_Node;
  with Current_Node^. Immediate_siblings^ do
    if Element = Previous_Node
      then if Next = Nil
             then Current_Node := Nil
             else Current_Node := Next^. Element
      else Current_Node := Element;
  Previous_Node := Temp_Node
end;
```

```
{-------------------------------------------------------}
{- Gets a sibling of a node.                           -}
{-------------------------------------------------------}
function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

begin
  NBOur_Of := Node^. Immediate_Siblings^. Element;
end;

{-------------------------------------------------------}
{- Gets the other (if any) sibling not returned by Nbour_of -}
{-------------------------------------------------------}
function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

begin
  if Node^. Immediate_Siblings^. Next = Nil
    then Other_NBour_of := Nil
    else Other_NBour_of := Node^. Immediate_Siblings^. Next^. Element
end;

{-------------------------------------------------------}
{- A Q node is disposed of.                            -}
{-------------------------------------------------------}
procedure Kill_Node (var Node : PQ_Node_Ptr);

begin
  with Node^. Immediate_Siblings^ do     { We need to delete the sibling list as well }
    begin
      if Next <> Nil
        then dispose (Next);
      dispose (Node^. Immediate_Siblings)
    end;
  dispose (Node);                        { Kill the node }
  Node := Nil                            { and reset its value }
end;


end.
```

```
{-------------------------------------------------------}
{- Demoucran, Malgrange and Pertuisel algorithm.       -}
{-------------------------------------------------------}
unit DMP_Algorithm;

interface

uses
  Planar_Defs,
  Planar_Miscellaneous,
  CRT;

procedure Demoucran_Malgrange_Pertuisel;    { main procedure }


{-------------------------------------------------------}
implementation

var
  Region_Vertices     : Array [Region_Range] of Vertex_List_Ptr;  { Stores vertices bounding a region }
  Number_in_H         : Vertex_Ptr_Range;                         { Number of vertices in Graph so far }
  Vertices_of_H       : Array [Vertex_Ptr] of Vertex_Ptr;         { The vertices added so far }
  Number_Frags_Found  : Integer;                                  { Total fragments found so far }
  Fragments           : Attachments;                              { The information used for each fragment }

{-------------------------------------------------------}
{- The generate fragments procedure.                   -}
{- See Chapter 2 for details.                          -}
{-------------------------------------------------------}
procedure Generate_Fragments;

var
  Current_Frag_Posn   : Integer;          { Position we are placing a new fragment into }
  Temp_Attach         : Attach_List_Ptr;  { Temporary attachment variable }
  Active_Vertex_Label : Integer;          { Labelling vertex variable }
  Current_Vertex_in_H : Vertex_Ptr_Range; { Current vertex we are generating fragments from }
  Root                : Attach_List_Ptr;  { Root attachment of current fragment }
  Next_Vertex,
  Active_Vertex       : Vertex_Ptr;       { Current vertex we are generating from }
  Start_Edge          : Edge_Ptr;         { temporary variable }
  Finished            : Boolean;          { Finished generating fragments }

{-------------------------------------------------------}
{- Initialisation Routine.                             -}
{-------------------------------------------------------}
procedure Initialise_Generate_Fragments;

var
  Temp_Edge   : Edge_Ptr;
  Temp_Vertex : Vertex_Ptr;

begin
  Temp_Vertex := 1;
  For Temp_Vertex := 1 to Last_Vertex do   { Reset all edge and vertex markers }
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while Temp_Edge <> Nil do
        begin
```

```
        Temp_Edge^. Frag_Used := Temp_Edge^. Used;   { Used if it is in the graph already }
        Temp_Edge := Temp_Edge^. Next
      end;
    Graph^ [Temp_Vertex]. Mark1 := 0;
  end;
 Active_Vertex_Label := 1;   { Starting vertex label }
 Current_Vertex_in_H := 0;   { Current vertex 0 }
 Current_Frag_Posn := 0;     { start placing at the beginning }
 Finished := false;          { not finished yet }
 Root := Nil                 { Current fragment is empty }
end;


{-------------------------------------------------------}
{- Creates a new fragment.                              -}
{-------------------------------------------------------}
procedure Create_New_Fragment;

begin
 if (Root = Nil) or (Root^. Next <> Nil)   { If Fragment not generated yet or not empty }
   then begin                              { then generate a new one }
     Active_Vertex := Vertices_of_H [Current_Vertex_in_H];  { Get start active vertex }
     Graph^ [Active_Vertex]. Father := 0;                   { no father since root }
     new (Root);
     with Root^ do
       begin
         Next := nil;                                       { no other attachments yet }
         Father := 0;                                       { at root - so no father }
         Vertex := Active_Vertex                            { and vertex is this one }
       end;
     Current_Frag_Posn := Current_Frag_Posn + 1;
     while (Current_Frag_Posn <= Number_Frags_Found)        { search for a position to place the fragment }
             and (Fragments [Current_Frag_Posn]. Attachments <> Nil) do
       Current_Frag_Posn := Current_Frag_Posn + 1;
     if Current_Frag_Posn > Number_Frags_Found              { check we don't exceed max }
       then Number_Frags_Found := Number_Frags_Found + 1;
     Fragments [Current_Frag_Posn]. Common_Regions := Nil;
     Fragments [Current_Frag_Posn]. Attachments := Root     { and save the Root }
   end
   else begin          { The fragment was generated but not used - so re-use }
     Active_Vertex := Vertices_of_H [Current_Vertex_in_H];  { new active vertex }
     Root^. Vertex := Active_Vertex;                        { and update root accordingly }
   end
end;
```

```
{-------------------------------------------------------}
{- Find the edge and vertex set of a fragment          -}
{- See Section 2.1 - we use a DFS.                     -}
{-------------------------------------------------------}
procedure Build_Fragment (Start_Vertex : Vertex_Ptr);

var
  Temp_Edge : Edge_Ptr;

begin
  Graph^ [Start_Vertex]. Mark1 := Active_Vertex_Label;   { note scanned }
  Active_Vertex_Label := Active_Vertex_Label + 1;
  if Graph^ [Start_Vertex]. Used              { if on the subgraph H already }
    then begin
      new (Temp_Attach);                      { add a new attachment }
      Temp_Attach^. Next := Root^. Next;      { to the list of attachments }
      Root^. Next := Temp_Attach;
      Temp_Attach^. Vertex := Start_Vertex;
      Temp_Attach^. Father := Graph^ [Start_Vertex]. Father;  { and backtrack after this }
    end
    else begin
      Temp_Edge := Graph^ [Start_Vertex]. Edges;  { check if nbour is unexplored }
      while Temp_Edge <> Nil do
        begin
          if not Temp_Edge^. Frag_Used         { if not used by the fragment }
            then begin
              Temp_Edge^. Frag_Used := true;   { add it to a fragment }
              Temp_Edge^. Other_Edge^. Frag_Used := true;
              Next_Vertex := Temp_Edge^. Vertex;
                                               { and check if nbour needs to be visited }
              if Graph^ [Next_Vertex]. Mark1 < Graph^ [Root^. Vertex]. Mark1
                then begin
                  Graph^ [Next_Vertex]. Father := Start_Vertex;
                  Build_Fragment (Next_Vertex) { then visit it }
                end
            end;
          Temp_Edge := Temp_Edge^. Next
        end
    end
end;
```

```
{-------------------------------------------------------------}
{- Main procedure for Generating fragments.                  -}
{- See Section 2.1                                           -}
{-------------------------------------------------------------}
begin
  Initialise_Generate_Fragments;
  while not Finished do
    begin
      {2}
      if Current_Vertex_in_H <= Number_in_H
        then begin
          repeat                { get a new vertex from H to be root }
            Current_Vertex_in_H := Current_Vertex_in_H + 1;
          until (Current_Vertex_in_H > Number_in_H)
               or (Graph^ [Vertices_of_H [Current_Vertex_in_H]]. Gen_Frags);
          if (Current_Vertex_in_H > Number_in_H)
            then Finished := true
            else begin                      { we have a vertex to search from }
              Active_Vertex := Vertices_of_H [Current_Vertex_in_H];
              Graph^ [Active_Vertex]. Father := 0;
              Graph^ [Active_Vertex]. Gen_Frags := false;
              Graph^ [Active_Vertex]. Mark1 := Active_Vertex_Label  { and give the vertex a new label }
            end
        end
        else Finished := true;
      if not Finished
        then begin
          Start_Edge := Graph^ [Active_Vertex]. Edges;   { check all edges incident }
          repeat
            if not Start_Edge^. Frag_Used               { if not part of a fragment }
              then begin
                Active_Vertex_Label := Active_Vertex_Label + 1;
                Graph^ [Active_Vertex]. Mark1 := Active_Vertex_Label;
                Active_Vertex_Label := Active_Vertex_Label + 1;
                Create_New_Fragment;                    { then we start a new fragment }
                Start_Edge^. Frag_Used := true;
                Start_Edge^. Other_Edge^. Frag_Used := true;
                Graph^ [Start_Edge^. Vertex]. Father := Active_Vertex;
                Build_Fragment (Start_Edge^. Vertex);    { and explore it recursively }
              end;
            Start_Edge := Start_Edge^. Next;     { check next edge }
          until Start_Edge = Nil
        end
    end;
  if (Root <> Nil) and (Root^. Next = Nil)      { the last fragment generated was an empty one        }
    then begin              { reclaim it and adjust the fragment table accordingly }
      dispose (Root);
      if Current_Frag_Posn = Number_Frags_Found      { and check for underflow }
        then Number_Frags_Found := Number_Frags_Found - 1;
      Fragments [Current_Frag_Posn]. Attachments := Nil
    end
end;
```

```
{-------------------------------------------------------}
{- Straight forward Depth First Search - see Hopcroft and   -}
{- Tarjan for details on the DFS mechanism.                 -}
{- We need to do a DFS to                                   -}
{-             i) Get 2 connected components                -}
{-             ii) Find the initial cycle efficiently       -}
{-------------------------------------------------------}
procedure DFS;

function Min (x, y : Integer) : Integer;

begin
  if x < y
    then Min := x
    else Min := y;
end;

var
  Current_Label : 0..MaxInt;
  Temp_Edge,
  Temp_Edge2  : Edge_Ptr;
  Finished,
  New_Vertex  : Boolean;
  Temp_Vertex : Vertex_Ptr;

begin
{Initialising all Fathers, Labels to 0 }
  For Temp_Vertex := 1 to Last_Vertex do
    with Graph^ [Temp_Vertex] do
      begin
        Number := 0;
        Father := 0;
        L1 := 0;
        Used := false;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do
          begin
            Temp_Edge^. Deleted := False;
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;
  New_Vertex := true;
  Temp_Vertex := 1;
  Current_Label := 0;
  repeat
    repeat
      {2}
      if New_Vertex
        then begin
          Current_Label := Current_Label + 1;
          Graph^ [Temp_Vertex]. Number := Current_Label;
          Graph^ [Temp_Vertex]. L1 := Current_Label;
        end;
      {3}
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      While (Temp_Edge <> Nil) and
              ((Temp_Edge^. Deleted) or (Temp_Edge^. Used)) do
```

```
        Temp_Edge := Temp_Edge^. Next;
      if (Temp_Edge <> Nil)
        then begin
          {4}
             { Direct the Edge }
            Temp_Edge^. Used := True;
            Temp_Edge^. Other_Edge^. Deleted := true;
            with Graph^ [Temp_Edge^. Vertex] do
              if Number <> 0
                then begin { Back Edge - adjust L1 and return }
                  if Number < Graph^ [Temp_Vertex]. L1
                    then begin
                      Graph^ [Temp_Vertex]. L1 := Number;
                    end
                    else;
                  New_Vertex := false;
                end
                else begin
                  Father := Temp_Vertex;
                  Temp_Vertex := Temp_Edge^. Vertex;
                  New_Vertex := true;
                end;
        end
    until (Temp_Edge = Nil);
    if Graph^ [Temp_Vertex]. Number = 1
      then
        {5} Finished := true
      else begin
        {6}
        with Graph^ [Graph^ [Temp_Vertex]. Father] do
          if Graph^ [Temp_Vertex]. L1 < L1
            then L1 := Graph^ [Temp_Vertex]. L1;
        Temp_Vertex := Graph^ [Temp_Vertex]. Father;
        New_Vertex := false;
      end
  until Finished;
end;
```

```
{-------------------------------------------------------}
{- Main algorithm procedure                             -}
{-------------------------------------- Full_Kids_Count := Full_Kids_Count + Partial_Child_2^. Full_Kids_
                end;
              Kill_Node (Partial_Child_1);    { cleanup memory allocations }
              if Partial_Child_2 <> Nil
                then begin
                  Kill_Node (Partial_Child_2);
                  dispose (Partial_Kids^. Next)
                end;
              dispose (Partial_Kids);
            end;
          Template_Q3 := true
        end;
end;

{-------------------------------------------------------}
{- Main Initialise Routine                              -}
{-------------------------------------------------------}
procedure Initialise_Reduction;

var
  Dummy_PQ     : PQ_Node_Ptr;

begin
  if Queue_Start <> Nil
    then repeat
      Remove_from_Queue (Dummy_PQ)     { Cleanup after Bubble Phase }
    until Queue_Start = Nil;
  Root_Node := Nil
end;
```

```
begin
  Initialise_Reduction;                          { Do main Init }
  Place_Leaves_On_Queue;                         { Place pertinent leaves on Queue }
  Sizeof_Set_S := Sizeof_Queue;
  while (Sizeof_Queue > 0) and (Planar) do
    begin
      Remove_from_Queue (Current_Node);
      if Current_Node^. Node_Type = Leaf
        then Current_Node^. Pert_Leaf_Count := 1;
      if Current_Node^. Pert_Leaf_Count < Sizeof_Set_S    { Check if we are at pertinent Root }
        then begin                        { if not, then }
          Current_Parent := Current_Node^. Parent;
          Current_Parent^. Pert_Leaf_Count :=     { update Parent's Pert_Leaf_Count of Pertinent Leaves }
            Current_Parent^. Pert_Leaf_Count + Current_Node^. Pert_Leaf_Count;
          Current_Parent^. Pert_Child_Count :=    { Update number of kids left to process }
            Current_Parent^. Pert_Child_Count - 1;
          if Current_Parent^. Pert_Child_Count = 0  { see if all kids matched }
            then Add_to_Queue (Current_Parent);
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P3 (Current_Node) then
          if not Template_P5 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
            begin
              writeln ('Halting no template matches');
              Planar := false
            end
        end
        else begin       { This node is the Pertinent Root }
          Root_Node := Current_Node;              { default is this node is Pertinent Root }
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P2 (Current_Node) then
          if not Template_P4 (Current_Node) then
          if not Template_P6 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
          if not Template_Q3 (Current_Node) then
            begin
              writeln ('Halting no template matches at pertinent root');
              Planar := false
            end
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit contains the routines use by both the Bubble   -}
{- and Reduction units of Lempel, Even and Cederbaum.       -}
{-------------------------------------------------------}
unit Lempel_Globals;

Interface

uses
  CRT,
  Planar_Miscellaneous,
  Planar_Defs;

{-------------------------------------------------------}
{- All the global variables.                            -}
{-------------------------------------------------------}
var
  ST_Number_Index    : Array [1..Max_Vertices] of Vertex_Ptr;  { used to map a St Number to }
                                                               { an actual vertex number    }
  Used_List          : List_Ptr;          { used to reset values after each reduction }
  Pseudo_Node,                            { see chapter 4 for details }
  New_Root,                               { the new root node to add new leaves to after a reduction }
  Root_Node          : PQ_Node_Ptr;       { the Root of the pertinent tree after a reduction }
  Sizeof_Queue       : Integer;           { Queue size for bubble and reduction }
  Planar             : Boolean;
  Current_Vertex,                         { Used for planarity testing of graph }
  Num_Vertices_Added : Vertex_Ptr;        { to the current component      }
  Queue_Start,                            { For the Queue                 }
  Queue_Head         : Double_Ptr;

procedure Kill_Node (var Node : PQ_Node_Ptr);               { wipe memory  allocation of Node }

procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr);  { walk between siblings }

function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;       { get an immediate sibling }

function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr; { get other immediate sibling }

procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);   { creates a new node with defaults }

procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);          { adds a node to the queue }

procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr); { removes a node from the queue }

procedure Place_Leaves_on_Queue;                            { adds all pertinent leaves to the queue }

procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;  { adds a child to a P node parent }
                              var Child_Ptr : PQ_Node_Ptr);

procedure Add_Edges_to_Tree (From_Vertex : Vertex_Ptr;     { adds new leaves to the tree }
                             Tree_Node   : PQ_Node_Ptr);

procedure Add_Replace_Sibling (New_Node, Old_Node, Other_Node : PQ_Node_Ptr);
                         { adds Other_Node to New_Node's Sibling list and }
                         { replaces Old_Node with New_Node in Other_Node's Sibling list }

procedure Get_Full_Empty_Siblings (var Node,
                                   Full_Sibling, Empty_Sibling : PQ_Node_Ptr);
```

```
                        { Gets the full/partial sibling and empty sibling of a node }

procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);   { adds a single node to a used list }

procedure Add_List_to_Used_List (var List_Start : List_Ptr);      { adds a list of nodes to used list }

procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);
                        { Replaces in the Parent all references to Old_Node by New_Node }

Implementation

{-------------------------------------------------------------}
{- Returns a PQ_Node initialised with default values         -}
{- Will not initialise fields dependent on node type.        -}
{-------------------------------------------------------------}
procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);

begin
 new (PQ_Pointer);
 with PQ_Pointer^ do                { set the initial values }
   begin
     Data_Label := Empty;
     Full_Kids := Nil;
     Partial_Kids := Nil;
     Immediate_Siblings := Nil;
     Mark := None;
     Parent := Nil;
     Child_Count := 0;
     Full_Kids_Count := 0;
     Pert_Child_Count := 0;
     Pert_Leaf_Count := 0;
   end
end;

{-------------------------------------------------------------}
{- Adds a PQ node to the Queue                               -}
{-------------------------------------------------------------}
procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);

var Queue_Element : Double_Ptr;

begin
  new (Queue_Element);
  Queue_Element^. Left := Nil;
  Queue_Element^. Right := Queue_Start;
  Queue_Element^. Element := PQ_Element;
  if Queue_Start <> Nil
    then Queue_Start^. Left := Queue_Element;
  Queue_Start := Queue_Element;
  if Queue_Head = Nil
    then Queue_Head := Queue_Element;
  Sizeof_Queue := Sizeof_Queue + 1
end;
```

```
{-------------------------------------------------------------}
{- Removes a PQ node from the Queue                          -}
{-------------------------------------------------------------}
procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr);

var  Queue_Element : Double_Ptr;

begin
  if Queue_Start = Nil
    then Halt;
  Queue_Element := Queue_Head;
  Queue_Head := Queue_Head^. Left;
  if Queue_Start^. Right = Nil
    then Queue_Start := Nil;
  if Queue_Head <> Nil
    then Queue_Head^. Right := Nil;
  PQ_Element := Queue_Element^. Element;
  dispose (Queue_Element);
  Sizeof_Queue := Sizeof_Queue - 1
end;

{-------------------------------------------------------------}
{- Adds all pertinent leaves to the Queue.                   -}
{-------------------------------------------------------------}
procedure Place_Leaves_on_Queue;

var Temp_Edge : Edge_Ptr;

begin
  Temp_Edge := Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges;  { get start edge }
  while (Temp_Edge <> Nil) do
    begin
      if (Graph^ [Temp_Edge^. Vertex]. ST_Number < Num_Vertices_Added)  { if it is part of the }
        then Add_to_Queue (Temp_Edge^. PQ_Ptr);                         { graph so far then add to queue }
      Temp_Edge := Temp_Edge^. Next
    end
end;
```

```
{-------------------------------------------------------}
{- Adds Child_Ptr to the circular list of Parent_Node.  -}
{-------------------------------------------------------}
procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;
                              var Child_Ptr : PQ_Node_Ptr);
var Temp_Double : Double_Ptr;

begin
  with Parent_Node^ do
    if List_Start = Nil           { check for initial case }
      then begin
        new (List_Start);
        with List_Start^ do       { set up single element circular list }
          begin
            Left := List_Start;
            Right := List_Start;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := List_Start
          end;
      end
      else begin
        new (Temp_Double);
        with Temp_Double^ do      { normal insertion into circular list }
          begin
            Left := List_Start^. Left;
            Right := List_Start;
            List_Start^. Left^. Right := Temp_Double;
            List_Start^. Left := Temp_Double;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := Temp_Double
          end
      end
end;
```

```
{-------------------------------------------------------}
{- Replace all references of Old_Node with New_Node in      -}
{- Parent. Replace it in Sibling Chains and Circular list if-}
{- necessary. Delete_Node indicates if New_Node is a child  -}
{- of Old_Node and whether it must be deleted from Old_Node -}
{- circular list.                                           -}
{- Also add New_Node to the partial children list of parent.-}
{-------------------------------------------------------}
procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);

var                              { all variables are temporary }
  Temp_Element2,
  List_Element   : List_Ptr;
  Double_Element : Double_Ptr;
  Temp_Node      : PQ_Node_Ptr;

begin
  New_Node^. Parent := Old_Node^. Parent;     { Get new parent }
  New_Node^. Pert_Leaf_Count := Old_Node^. Pert_Leaf_Count;
  New_Node^. Data_Label := Partial;        { new node is partial }
  if Root_Node = Nil                       { so add to Partial children list }
    then begin
      new (List_Element);
      List_Element^. Next := New_Node^. Parent^. Partial_Kids;
      List_Element^. Element := New_Node;
      New_Node^. Parent^. Partial_Kids := List_Element
    end;
  if Delete_Node      { We must delete Old_Node from circular list of Old_Node }
    then begin
      Double_Element := New_Node^. Circ_List_Posn;
      if Double_Element^. Left = Double_Element    { check for end condition }
        then Old_Node^. List_Start := Nil
        else begin                                 { otherwise delete as normal }
          if Double_Element = Old_Node^. List_Start
            then Old_Node^. List_Start := Double_Element^. Right;
          Double_Element^. Left^. Right := Double_Element^. Right;
          Double_Element^. Right^. Left := Double_Element^. Left;
        end;
      New_Node^. Circ_List_Posn := Nil;
      Old_Node^. Child_Count := Old_Node^. Child_Count - 1;  { note a child has been removed }
      dispose (Double_Element);
    end;
  if Old_Node^. Immediate_Siblings = Nil    { then Parent is a P Node and replace }
    then begin                              { in parent's circulra list.          }
      New_Node^. Immediate_Siblings := Nil;
      New_Node^. Circ_List_Posn := Old_Node^. Circ_List_Posn;
      New_Node^. Circ_List_Posn^. Element := New_Node;
    end
    else begin                              { Replace in immediate sibling's Siblings chain }
      New_Node^. Immediate_Siblings :=
           Old_Node^. Immediate_Siblings;
      Old_Node^. Immediate_Siblings := Nil;
      List_Element := New_Node^. Immediate_Siblings;   { with each sibling }
      while (List_Element <> Nil) do
        begin
          Temp_Node := List_Element^. Element;
          Temp_Element2 := Temp_Node^. Immediate_Siblings;    { search fro refernce to Old_Node }
```

```
          while (Temp_Element2^. Element <> Old_Node) do
            Temp_Element2 := Temp_Element2^. Next;
          Temp_Element2^. Element := New_Node;                { replace with New_Node }
          List_Element := List_Element^. Next          { and go to next sibling }
        end;
      if New_Node^. Parent^. LeftMost_Kid = Old_Node          { adjust end most pointers as well }
        then New_Node^. Parent^. LeftMost_Kid := New_Node
        else if New_Node^. Parent^. RightMost_Kid = Old_Node
              then New_Node^. Parent^. RightMost_Kid := New_Node;
    end;
end;

{-------------------------------------------------------------}
{- Add new leaves representing edges from From_Vertex.      -}
{-------------------------------------------------------------}
procedure Add_Edges_to_Tree (From_Vertex : Vertex_Ptr;
                             Tree_Node   : PQ_Node_Ptr);

var
  New_Nodes       : Integer;            { used to check for only a single leaf }
                                        { added since we don't want chains     }
  Last_Edge,
  From_Temp_Edge  : Edge_Ptr;
  Current_ST      : Vertex_Ptr_Range;
  PQ_Element      : PQ_Node_Ptr;

begin
  New_Nodes := 0;
  From_Temp_Edge := Graph^ [From_Vertex]. Edges;
  Current_ST := Graph^ [From_Vertex]. ST_Number;
  while From_Temp_Edge <> Nil do                  { with each edge do }
    begin
      if Graph^ [From_Temp_Edge^. Vertex]. ST_Number > Current_ST   { check it goes to a higher vertex }
        then begin
          Create_PQ_Node (PQ_Element);          { create the new leaf }
          New_Nodes := New_Nodes + 1;
          with PQ_Element^ do                   { mark it as a leaf }
            begin
              Node_Type := Leaf;
              Parent := Tree_Node;
              Tail_Vertex := From_Vertex
            end;
          Tree_Node^. Child_Count := Tree_Node^. Child_Count + 1;   { add to Tree Node's kids }
          Add_to_Circle_Link (Tree_Node, PQ_Element);
          From_Temp_Edge^. PQ_Ptr := PQ_Element;              { and note which PQ Element reps this edge }
          Last_Edge := From_Temp_Edge;
          From_Temp_Edge^. Other_Edge^. PQ_Ptr := PQ_Element   { ditto for other edge element }
        end;
      From_Temp_Edge := From_Temp_Edge^. Next    { go to the next candidate edge }
    end;
  if New_Nodes = 0       { Error - ST Numbering guarantees we always can add an edge }
    then begin
      writeln ('Fatal error - no new edges to add');
      halt
    end
    else
      if New_Nodes = 1              { check for chains of a single element added }
        then with Tree_Node^ do                 { tree node becomes the leaf }
```

```
          begin
            PQ_Element := List_Start^. Element;
            dispose (List_Start);                { wipe the reference to the leaf }
            Node_Type := Leaf;                   { note this vertex is a leaf }
            Last_Edge^. PQ_Ptr := Tree_Node;       { change references to this node }
            Last_Edge^. Other_Edge^. PQ_Ptr := Tree_Node;
            Tail_Vertex := PQ_Element^. Tail_Vertex;
            dispose (PQ_Element)                 { and reclaim the node }
          end
end;

{-------------------------------------------------------------}
{- Adds Other_Node to New_Node's Sibling List.               -}
{- Also Replaces reference in Other_Node's Sibling list to   -}
{- Old_Node with New_Node.                                   -}
{-------------------------------------------------------------}
procedure Add_Replace_Sibling (New_Node, Old_Node,
                               Other_Node : PQ_Node_Ptr);
var
  List_Element : List_Ptr;

begin
  List_Element := Other_Node^. Immediate_Siblings;    { Replace phase }
  if List_Element^. Element = Old_Node
    then List_Element^. Element := New_Node
    else List_Element^. Next^. Element := New_Node;
  new (List_Element);
  List_Element^. Next := New_Node^. Immediate_Siblings;   { add reference to Other_Node }
  List_Element^. Element := Other_Node;
  New_Node^. Immediate_Siblings := List_Element
end;
```

```
{-------------------------------------------------------}
{- The Full and Empty Siblings of a node are returned.      -}
{- For the purposes of this routine, a full node is either  -}
{- strictly full, or partial.                               -}
{-------------------------------------------------------}
procedure Get_Full_Empty_Siblings (var Node, Full_Sibling,
                                           Empty_Sibling : PQ_Node_Ptr);

var
  Temp,
  Temp2 : PQ_Node_Ptr;

begin
  if Node <> Nil
    then begin
      Temp := Node^.Immediate_Siblings^.Element;
      if Node^. Immediate_Siblings^. Next = Nil
        then Temp2 := nil
        else Temp2 := Node^. Immediate_Siblings^. Next^. Element;
      if Temp^. Data_Label <> Empty
        then begin
          Full_Sibling := Temp;
          Empty_Sibling := Temp2
        end
        else begin
          Empty_Sibling := Temp;
          Full_Sibling := Temp2
        end
    end
    else begin
      Empty_Sibling := Nil;
      Full_Sibling := Nil
    end
end;

{-------------------------------------------------------}
{- If one of the Endmost kids was Old_Node, then we reset   -}
{- that kid to be New_Kid.                                  -}
{-------------------------------------------------------}
procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

begin
  with Current_Node^ do
    if RightMost_Kid = Old_Kid
      then begin
        RightMost_Kid := New_Kid;
        New_Kid^. Parent := Current_Node;
      end
      else
        if LeftMost_Kid = Old_Kid
          then begin
            LeftMost_Kid := New_Kid;
            New_Kid^. Parent := Current_Node
          end
end;
```

```
{-------------------------------------------------------}
{-  A List is added to the Used List.                  -}
{-------------------------------------------------------}
procedure Add_List_to_Used_List (var List_Start : List_Ptr);

var
  List_Element : List_Ptr;

begin
  if List_Start <> Nil
    then begin
      List_Element := List_Start;
      while List_Element^. Next <> Nil do     { go to the end of the list }
        List_Element := List_Element^. Next;
      List_Element^. Next := Used_List;       { Append the list to the front of Used list }
      Used_List := List_Start;
      List_Start := Nil
    end
end;

{-------------------------------------------------------}
{- A single node is appended to used list.             -}
{-------------------------------------------------------}
procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Next := Used_List;
  List_Element^. Element := PQ_Element;
  Used_List := List_Element
end;

{-------------------------------------------------------}
{- We walk from Current_Node to an immediate sibling that is-}
{- not Previous_Node.                                  -}
{-------------------------------------------------------}
procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr);

var
  Temp_Node : PQ_Node_Ptr;

begin
  Temp_Node := Current_Node;
  with Current_Node^. Immediate_siblings^ do
    if Element = Previous_Node
      then if Next = Nil
             then Current_Node := Nil
             else Current_Node := Next^. Element
      else Current_Node := Element;
  Previous_Node := Temp_Node
end;
```

```
{-------------------------------------------------------------}
{- Gets a sibling of a node.                                  -}
{-------------------------------------------------------------}
function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

begin
  NBOur_Of := Node^. Immediate_Siblings^. Element;
end;

{-------------------------------------------------------------}
{- Gets the other (if any) sibling not returned by Nbour_of -}
{-------------------------------------------------------------}
function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

begin
  if Node^. Immediate_Siblings^. Next = Nil
    then Other_NBour_of := Nil
    else Other_NBour_of := Node^. Immediate_Siblings^. Next^. Element
end;

{-------------------------------------------------------------}
{- A Q node is disposed of.                                   -}
{-------------------------------------------------------------}
procedure Kill_Node (var Node : PQ_Node_Ptr);

begin
  with Node^. Immediate_Siblings^ do     { We need to delete the sibling list as well }
    begin
      if Next <> Nil
        then dispose (Next);
      dispose (Node^. Immediate_Siblings)
    end;
  dispose (Node);                        { Kill the node }
  Node := Nil                            { and reset its value }
end;


end.
```

```
{-------------------------------------------------------}
{- Demoucran, Malgrange and Pertuisel algorithm.       -}
{-------------------------------------------------------}
unit DMP_Algorithm;

interface

uses
  Planar_Defs,
  Planar_Miscellaneous,
  CRT;

procedure Demoucran_Malgrange_Pertuisel;    { main procedure }


{-------------------------------------------------------}
implementation

var
  Region_Vertices     : Array [Region_Range] of Vertex_List_Ptr;  { Stores vertices bounding a region }
  Number_in_H         : Vertex_Ptr_Range;                         { Number of vertices in Graph so far }
  Vertices_of_H       : Array [Vertex_Ptr] of Vertex_Ptr;         { The vertices added so far }
  Number_Frags_Found  : Integer;                                  { Total fragments found so far }
  Fragments           : Attachments;                              { The information used for each fragment }

{-------------------------------------------------------}
{- The generate fragments procedure.                   -}
{- See Chapter 2 for details.                          -}
{-------------------------------------------------------}
procedure Generate_Fragments;

var
  Current_Frag_Posn   : Integer;          { Position we are placing a new fragment into }
  Temp_Attach         : Attach_List_Ptr;  { Temporary attachment variable }
  Active_Vertex_Label : Integer;          { Labelling vertex variable }
  Current_Vertex_in_H : Vertex_Ptr_Range; { Current vertex we are generating fragments from }
  Root                : Attach_List_Ptr;  { Root attachment of current fragment }
  Next_Vertex,
  Active_Vertex       : Vertex_Ptr;       { Current vertex we are generating from }
  Start_Edge          : Edge_Ptr;         { temporary variable }
  Finished            : Boolean;          { Finished generating fragments }

{-------------------------------------------------------}
{- Initialisation Routine.                             -}
{-------------------------------------------------------}
procedure Initialise_Generate_Fragments;

var
  Temp_Edge   : Edge_Ptr;
  Temp_Vertex : Vertex_Ptr;

begin
  Temp_Vertex := 1;
  For Temp_Vertex := 1 to Last_Vertex do   { Reset all edge and vertex markers }
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while Temp_Edge <> Nil do
        begin
```

```
        Temp_Edge^. Frag_Used := Temp_Edge^. Used;   { Used if it is in the graph already }
        Temp_Edge := Temp_Edge^. Next
      end;
    Graph^ [Temp_Vertex]. Mark1 := 0;
  end;
 Active_Vertex_Label := 1;   { Starting vertex label }
 Current_Vertex_in_H := 0;   { Current vertex 0 }
 Current_Frag_Posn := 0;     { start placing at the beginning }
 Finished := false;          { not finished yet }
 Root := Nil                 { Current fragment is empty }
end;


{-------------------------------------------------------}
{- Creates a new fragment.                              -}
{-------------------------------------------------------}
procedure Create_New_Fragment;

begin
  if (Root = Nil) or (Root^. Next <> Nil)   { If Fragment not generated yet or not empty }
    then begin                              { then generate a new one }
      Active_Vertex := Vertices_of_H [Current_Vertex_in_H];  { Get start active vertex }
      Graph^ [Active_Vertex]. Father := 0;                   { no father since root }
      new (Root);
      with Root^ do
        begin
          Next := nil;                                       { no other attachments yet }
          Father := 0;                                       { at root - so no father }
          Vertex := Active_Vertex                            { and vertex is this one }
        end;
      Current_Frag_Posn := Current_Frag_Posn + 1;
      while (Current_Frag_Posn <= Number_Frags_Found)        { search for a position to place the fragment }
              and (Fragments [Current_Frag_Posn]. Attachments <> Nil) do
        Current_Frag_Posn := Current_Frag_Posn + 1;
      if Current_Frag_Posn > Number_Frags_Found              { check we don't exceed max }
        then Number_Frags_Found := Number_Frags_Found + 1;
      Fragments [Current_Frag_Posn]. Common_Regions := Nil;
      Fragments [Current_Frag_Posn]. Attachments := Root     { and save the Root }
    end
    else begin          { The fragment was generated but not used - so re-use }
      Active_Vertex := Vertices_of_H [Current_Vertex_in_H];  { new active vertex }
      Root^. Vertex := Active_Vertex;                        { and update root accordingly }
    end
end;
```

```
{-------------------------------------------------------}
{- Find the edge and vertex set of a fragment          -}
{- See Section 2.1 - we use a DFS.                     -}
{-------------------------------------------------------}
procedure Build_Fragment (Start_Vertex : Vertex_Ptr);

var
  Temp_Edge : Edge_Ptr;

begin
  Graph^ [Start_Vertex]. Mark1 := Active_Vertex_Label;   { note scanned }
  Active_Vertex_Label := Active_Vertex_Label + 1;
  if Graph^ [Start_Vertex]. Used              { if on the subgraph H already }
    then begin
      new (Temp_Attach);                      { add a new attachment }
      Temp_Attach^. Next := Root^. Next;      { to the list of attachments }
      Root^. Next := Temp_Attach;
      Temp_Attach^. Vertex := Start_Vertex;
      Temp_Attach^. Father := Graph^ [Start_Vertex]. Father;  { and backtrack after this }
    end
    else begin
      Temp_Edge := Graph^ [Start_Vertex]. Edges;  { check if nbour is unexplored }
      while Temp_Edge <> Nil do
        begin
          if not Temp_Edge^. Frag_Used         { if not used by the fragment }
            then begin
              Temp_Edge^. Frag_Used := true;   { add it to a fragment }
              Temp_Edge^. Other_Edge^. Frag_Used := true;
              Next_Vertex := Temp_Edge^. Vertex;
                                               { and check if nbour needs to be visited }
              if Graph^ [Next_Vertex]. Mark1 < Graph^ [Root^. Vertex]. Mark1
                then begin
                  Graph^ [Next_Vertex]. Father := Start_Vertex;
                  Build_Fragment (Next_Vertex) { then visit it }
                end
            end;
          Temp_Edge := Temp_Edge^. Next
        end
    end
end;
```

```
{-------------------------------------------------------}
{- Main procedure for Generating fragments.            -}
{- See Section 2.1                                     -}
{-------------------------------------------------------}
begin
  Initialise_Generate_Fragments;
  while not Finished do
    begin
      {2}
      if Current_Vertex_in_H <= Number_in_H
        then begin
          repeat                { get a new vertex from H to be root }
            Current_Vertex_in_H := Current_Vertex_in_H + 1;
          until (Current_Vertex_in_H > Number_in_H)
               or (Graph^ [Vertices_of_H [Current_Vertex_in_H]]. Gen_Frags);
          if (Current_Vertex_in_H > Number_in_H)
            then Finished := true
            else begin                      { we have a vertex to search from }
              Active_Vertex := Vertices_of_H [Current_Vertex_in_H];
              Graph^ [Active_Vertex]. Father := 0;
              Graph^ [Active_Vertex]. Gen_Frags := false;
              Graph^ [Active_Vertex]. Mark1 := Active_Vertex_Label  { and give the vertex a new label }
            end
        end
        else Finished := true;
      if not Finished
        then begin
          Start_Edge := Graph^ [Active_Vertex]. Edges;   { check all edges incident }
          repeat
            if not Start_Edge^. Frag_Used               { if not part of a fragment }
              then begin
                Active_Vertex_Label := Active_Vertex_Label + 1;
                Graph^ [Active_Vertex]. Mark1 := Active_Vertex_Label;
                Active_Vertex_Label := Active_Vertex_Label + 1;
                Create_New_Fragment;                    { then we start a new fragment }
                Start_Edge^. Frag_Used := true;
                Start_Edge^. Other_Edge^. Frag_Used := true;
                Graph^ [Start_Edge^. Vertex]. Father := Active_Vertex;
                Build_Fragment (Start_Edge^. Vertex);   { and explore it recursively }
              end;
            Start_Edge := Start_Edge^. Next;     { check next edge }
          until Start_Edge = Nil
        end
    end;
  if (Root <> Nil) and (Root^. Next = Nil)      { the last fragment generated was an empty one          }
    then begin              { reclaim it and adjust the fragment table accordingly }
      dispose (Root);
      if Current_Frag_Posn = Number_Frags_Found      { and check for underflow }
        then Number_Frags_Found := Number_Frags_Found - 1;
      Fragments [Current_Frag_Posn]. Attachments := Nil
    end
end;
```

```
{-------------------------------------------------------}
{- Straight forward Depth First Search - see Hopcroft and  -}
{- Tarjan for details on the DFS mechanism.                -}
{- We need to do a DFS to                                  -}
{-             i) Get 2 connected components               -}
{-             ii) Find the initial cycle efficiently      -}
{-------------------------------------------------------}
procedure DFS;

function Min (x, y : Integer) : Integer;

begin
  if x < y
    then Min := x
    else Min := y;
end;

var
  Current_Label : 0..MaxInt;
  Temp_Edge,
  Temp_Edge2  : Edge_Ptr;
  Finished,
  New_Vertex  : Boolean;
  Temp_Vertex : Vertex_Ptr;

begin
{Initialising all Fathers, Labels to 0 }
  For Temp_Vertex := 1 to Last_Vertex do
    with Graph^ [Temp_Vertex] do
      begin
        Number := 0;
        Father := 0;
        L1 := 0;
        Used := false;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do
          begin
            Temp_Edge^. Deleted := False;
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;
  New_Vertex := true;
  Temp_Vertex := 1;
  Current_Label := 0;
  repeat
    repeat
      {2}
      if New_Vertex
        then begin
          Current_Label := Current_Label + 1;
          Graph^ [Temp_Vertex]. Number := Current_Label;
          Graph^ [Temp_Vertex]. L1 := Current_Label;
        end;
      {3}
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      While (Temp_Edge <> Nil) and
              ((Temp_Edge^. Deleted) or (Temp_Edge^. Used)) do
```

```
        Temp_Edge := Temp_Edge^. Next;
      if (Temp_Edge <> Nil)
        then begin
          {4}
             { Direct the Edge }
            Temp_Edge^. Used := True;
            Temp_Edge^. Other_Edge^. Deleted := true;
            with Graph^ [Temp_Edge^. Vertex] do
              if Number <> 0
                then begin { Back Edge - adjust L1 and return }
                  if Number < Graph^ [Temp_Vertex]. L1
                    then begin
                      Graph^ [Temp_Vertex]. L1 := Number;
                    end
                    else;
                  New_Vertex := false;
                end
                else begin
                  Father := Temp_Vertex;
                  Temp_Vertex := Temp_Edge^. Vertex;
                  New_Vertex := true;
                end;
        end
    until (Temp_Edge = Nil);
    if Graph^ [Temp_Vertex]. Number = 1
      then
        {5} Finished := true
      else begin
        {6}
        with Graph^ [Graph^ [Temp_Vertex]. Father] do
          if Graph^ [Temp_Vertex]. L1 < L1
            then L1 := Graph^ [Temp_Vertex]. L1;
        Temp_Vertex := Graph^ [Temp_Vertex]. Father;
        New_Vertex := false;
      end
  until Finished;
end;
```

```
{-------------------------------------------------------}
{- Main algorithm procedure                            -}
{-------------------------------------------------------}
procedure Demoucran_Malgrange_Pertuisel;

var
  Current_Vertex  : Vertex_Ptr_Range; { Used to detect a group of unused vertices - a component }
  Regions_Used,                       { The total number of Regions in the Graph so far }
  Region_to_Use,                      { The region that the chosen path must be embedded in }
  Fragment_to_Use : Region_Range;     { The Fragment position in the fragment array to use }
  Fragments_Generated,                { Notes if any new fragments have been generated }
  Planar,
  Finished_Component,                 { Finished testing the current component }
  Finished        : Boolean;
  Edges_Used,                         { Total edges embedded so far }
  Max_Edges       : Integer;          { Total number of edges in the Graph }

{-------------------------------------------------------}
{- DMP Initialise routine                              -}
{-------------------------------------------------------}
procedure Initialise_DMP;

var
  Loop        : Region_Range;
  Temp_Edge   : Edge_Ptr;
  Temp_Vertex : Vertex_Ptr;

begin
  Clrscr;
  DFS;
  if Check_2_Connected
    then begin
      Current_Vertex := 1;
      Max_Edges := 0;
      Edges_Used := 0;
      For Temp_Vertex := 1 to Last_Vertex do     { For each vertex initialise }
        begin
          Graph^ [Temp_Vertex]. Regions := Nil;    { no regions }
          Graph^ [Temp_Vertex]. Used := false;     { not added to the graph yet }
          Graph^ [Temp_Vertex]. Mark := 0;
          Temp_Edge := Graph^ [Temp_Vertex]. Edges;
          while Temp_Edge <> Nil do              { For each edge do }
            begin
              Max_Edges := Max_Edges + 1;          { count the edge }
              Temp_Edge^. Used := false;           { not added to the graph yet }
              Temp_Edge := Temp_Edge^. Next
            end;
        end;
      Max_Edges := Max_Edges div 2;          { we counted the edges twice }
      For Loop := 1 to Max_Edges div 3 do    { Initialise all the Regions to be empty }
        Region_Vertices [Loop] := Nil;
      if Max_Edges = 3
        then Region_Vertices [2] := Nil;
      Finished := false;
      Number_Frags_Found := 0;       { No fragments found }
      Number_In_H := 0;              { Nothing embedded yet }
      Planar := true
    end
```

```
    else Planar := false
end;

{-------------------------------------------------------}
{- Function checks if there are any fragments left to embed.-}
{- Used to detect the end of a component embedding.        -}
{-------------------------------------------------------}
function Check_Fragments_Left : Boolean;

var
  Temp : Integer;

begin
  Temp := 1;
  while (Temp <= Number_Frags_Found) and (Fragments [Temp]. Attachments = Nil) do  { all fragments must be n
    Temp := Temp + 1;                                                              { for component to be fin
  Check_Fragments_Left := not (Temp > Number_Frags_Found)
end;

{-------------------------------------------------------}
{- A vertex is now bounded by two new regions. The regions  -}
{- are the first two regions for this vertex - i.e. the     -}
{- vertex is new to the graph.                              -}
{- Note :- Region1 < Region2                                -}
{- Adjust_Region_List specifies if we must add the vertex   -}
{- to the corresponding Region lists or not.                -}
{-------------------------------------------------------}
procedure Add_Two_Regions_to_Vertex (Region1, Region2   : Region_Range;
                                     Vertex_Number      : Vertex_Ptr;
                                     Adjust_Region_List : Boolean);
var
  Region_Ptr     : Region_List_Ptr;
  Region_Element : Vertex_List_Ptr;

begin
  if Adjust_Region_List
    then begin
      new (Region_Element);
      Region_Element^. Vertex := Vertex_Number;
      Region_Element^. Next := Region_Vertices [Region1];
      Region_Vertices [Region1] := Region_Element;        { add the vertex to the vertex list }
      new (Region_Element);
      Region_Element^. Vertex := Vertex_Number;
      Region_Element^. Next := Region_Vertices [Region2]; { and the same for the other region }
      Region_Vertices [Region2] := Region_Element;
    end;
  new (Region_Ptr);                                       { now add the region to the vertex's list }
  Region_Ptr^. Region := Region2;                         { we add Region2 first because we need    }
  Region_Ptr^. Next := Graph^ [Vertex_Number]. Regions;   { to keep the vertex's region list sorted }
  Graph^ [Vertex_Number]. Regions := Region_Ptr;          { and Region1 < Region2.                  }
  new (Region_Ptr);
  Region_Ptr^. Region := Region1;
  Region_Ptr^. Next := Graph^ [Vertex_Number]. Regions;
  Graph^ [Vertex_Number]. Regions := Region_Ptr;
end;
```

```
{-------------------------------------------------------}
{- This procedure deletes Region_Del and adds Region_Add to -}
{- the vertex Vertex_Number.                              -}
{- Region_Add will always be the biggest in the sorted list.-}
{-------------------------------------------------------}
procedure Add_Delete_to_Regions (Region_Del, Region_Add : Region_Range;
                                 Vertex_Number          : Vertex_Ptr);
var
  Temp_Region,
  Region_Ptr,
  Region_Ptr2 : Region_List_Ptr;

begin
  new (Region_Ptr);
  Region_Ptr2 := Region_Ptr;
  Region_Ptr^. Region := Region_Add;
  Region_Ptr^. Next := Graph^ [Vertex_Number]. Regions;   { add the "dummy" onto the head of the }
  while (Region_Ptr^. Next <> Nil)                        { list to facilitate easy deletion.    }
           and (Region_Ptr^. Next^. Region <> Region_Del) do
    Region_Ptr := Region_Ptr^. Next;
  if Graph^ [Vertex_Number]. Regions = Region_Ptr^. Next
    then Graph^ [Vertex_Number]. Regions := Region_Ptr^. Next^. Next;
  Temp_Region := Region_Ptr^. Next;
  Region_Ptr^. Next := Region_Ptr^. Next^. Next;          { delete from the list }
  dispose (Temp_Region);                                  { and reclaim the memory }
  while Region_Ptr^. Next <> Nil do                       { move to the end of the list }
    Region_Ptr := Region_Ptr^. Next;
  Region_Ptr^. Next := Region_Ptr2;                       { and add Region_Add }
  Region_Ptr2^. Next := Nil
end;

{-------------------------------------------------------}
{- A Region is added to a vertex's region list.          -}
{-------------------------------------------------------}
procedure Add_Region_to_Vertex (Region_Add : Region_Range;
                                Vertex_Num : Vertex_Ptr);
var
  Region_Ptr,
  Region_Ptr2 : Region_List_Ptr;
  Temp_Vertex : Vertex_Ptr;

begin
  new (Region_Ptr);
  Region_Ptr2 := Region_Ptr;
  Region_Ptr^. Next := Graph^ [Vertex_Num]. Regions;    { add the dummy to the head of the }
  Region_Ptr^. Region := Region_Add;                    { list to facilitate easy insertion }
  while (Region_Ptr^. Next <> Nil) and
          (Region_Ptr^. Next^. Region < Region_Add) do
    Region_Ptr := Region_Ptr^. Next;
  if Region_Ptr <> Region_Ptr2
    then begin
      Region_Ptr2^. Next := Region_Ptr^. Next;          { normal insertion }
      Region_Ptr^. Next := Region_Ptr2
    end
    else Graph^ [Vertex_Num]. Regions := Region_Ptr2    { special case for inserting at the head }
end;
```

```
{-------------------------------------------------------}
{- The initial cycle is found.                         -}
{-------------------------------------------------------}
procedure Find_a_Cycle;

var
  Start_Vertex : Vertex_Ptr;
  Temp_Edge    : Edge_Ptr;
  Finished     : Boolean;

begin
  Start_Vertex := 1;     { the current vertex we are on }
  Finished := false;
  Regions_Used := 2;     { for each component we start with a cycle - 2 regions }
  while not Finished do  { until we complete the cycle }
    begin
      Graph^ [Start_Vertex]. Used := true;              { add this vertex to the graph }
      Graph^ [Start_Vertex]. Gen_Frags := true;         { generate fragments for this vertex }
      Add_two_Regions_to_Vertex (1, 2, Start_Vertex, true);   { and add the regions to the vertex's list }
      Number_in_H := Number_in_H + 1;                          { and add the vertex to the regions' lists }
      Vertices_of_H [Number_In_H] := Start_Vertex;      { Note the vertex we added to H }
      Temp_Edge := Graph^ [Start_Vertex]. Edges;
      while ((Graph^ [Temp_Edge^. Vertex]. L1 <> 1)        { Find the edge which has   }
                or (Graph^ [Temp_Edge^. Vertex]. Number <
                      Graph^ [Start_Vertex]. Number))     { destination vertex L1 = 1 }
            and (Temp_Edge^. Vertex <> 1) do
        Temp_Edge := Temp_Edge^. Next;
      Temp_Edge^. Used := true;
      Temp_Edge^. Other_Edge^. Used := true;            { the edge is now added }
      Start_Vertex := Temp_Edge^. Vertex;               { move to that vertex }
      Edges_Used := Edges_Used + 1;                     { another edge has been embedded }
      Finished := (Start_Vertex = 1)                    { Finished when cycle complete }
    end
end;
```

```
{-------------------------------------------------------}
{- The Common Regions for every fragment are determined.    -}
{- We keep track of the fragment with the lowest number of  -}
{- Common Regions, and if there is a fragment with no common-}
{- region between its attachments, then G is nonplanar.     -}
{-------------------------------------------------------}
procedure Determine_Common_Regions;

var
  Vertex1,                          { The first two attachments of a Fragment }
  Vertex2,
  Best_Fragment  : Vertex_Ptr;      { Number of the best fragment found so far }
  Temp_Regions,                     { temporary variables for adjusting region lists }
  Temp_Region2,
  Regions_Vertex1,                  { To access the region lists of the two attachments }
  Regions_Vertex2 : Region_List_Ptr;
  Current_Attach  : Attach_List_Ptr;  { The current attachment we are testing }
  Loop,                             { To access the fragments }
  Best_Count      : Integer;        { The number of Common Regions found for Best_Fragment }


{-------------------------------------------------------}
{- The first two attachments have been compared, and a list -}
{- of common regions has been created. Now we find the      -}
{- intersection of this common list with all the attachments-}
{- common lists.                                            -}
{-------------------------------------------------------}
procedure Check_Rest_of_Attachments;

begin
  Finished := (Current_Attach = Nil);       { check if there was only 2 attachments }
  new (Temp_Regions);
  Temp_Regions^. Next := Fragments [Loop]. Common_Regions;
  Fragments [Loop]. Common_Regions := Temp_Regions;         { add dummy to head of list to facilitate deletic
  while not Finished do
    begin
      Regions_Vertex1 :=                { start of the new attachments regions }
          Graph^ [Current_Attach^. Vertex]. Regions;
      Temp_Regions := Fragments [Loop]. Common_Regions;   { start of the common regions so far }
      while (Regions_Vertex1 <> Nil)
            and (Temp_Regions^. Next <> Nil)    { while we have not compared all }
            and (Planar) do
        begin
          while
             (Regions_Vertex1^. Region < Temp_Regions^. Next^. Region)
                and (Regions_Vertex1 <> Nil) do
            Regions_Vertex1 := Regions_Vertex1^. Next;   { search until attachment's  }
          if Regions_Vertex1 <> Nil                      { region is >= common region }
            then
              if (Temp_Regions <> Nil)
                   and (Temp_Regions^. Next^. Region     { check if they are equal }
                            = Regions_Vertex1^. Region)
                then begin
                  Temp_Regions := Temp_Regions^. Next;        { if so, then it is still common }
                  Regions_Vertex1 := Regions_Vertex1^. Next
                end
                else begin                                 { the region is no longer common }
                  Temp_Region2 := Temp_Regions^. Next;
```

```
                Temp_Regions^. Next := Temp_Regions^. Next^. Next;   { so delete it }
                dispose (Temp_Region2);
                Fragments [Loop]. Common_Count := Fragments [Loop]. Common_Count - 1
              end;
          if Fragments [Loop]. Common_Count = 0        { check if the fragment is still valid }
            then Planar := false
        end;                      { repeat until all regions compared }
      if Temp_Regions^. Next <> Nil
        then
          while Temp_Regions^. Next <> Nil do
            begin
              Temp_Region2 := Temp_Regions^. Next;
              Temp_Regions^. Next := Temp_Region2^. Next;
              dispose (Temp_Region2);
              Fragments [Loop]. Common_Count := Fragments [Loop]. Common_Count - 1
            end;
      Current_Attach := Current_Attach^. Next;   { get a new attachment }
      Finished := (not Planar) or (Current_Attach = Nil)
    end;
  Temp_Regions := Fragments [Loop]. Common_Regions;      { now get rid of dummy header }
  Fragments [Loop]. Common_Regions := Fragments [Loop]. Common_Regions^. Next;
  dispose (Temp_Regions);
end;

{-------------------------------------------------------}
{- The main Determine_Common_Regions procedure          -}
{-------------------------------------------------------}
begin
  Best_Count := Max_Edges;       { an impossible best case }
  Loop := 0;
  while (Loop < Number_Frags_Found) and (Planar) do    { while we have not tested all fragments }
    begin
      Loop := Loop + 1;
      Current_Attach := Fragments [Loop]. Attachments;  { Get the first attachment }
      if (Current_Attach <> Nil)   { Check that it is not empty }
        then begin
          if (Fragments [Loop]. Common_Regions = Nil)   { check if we need to re-generate the fragments }
            then begin                                  { if so, then }
              Vertex1 := Current_Attach^. Vertex;          { Get the first attachment }
              Current_Attach := Current_Attach^. Next;
              Vertex2 := Current_Attach^. Vertex;          { and the second attachment }
              Current_Attach := Current_Attach^. Next;
              Regions_Vertex1 := Graph^ [Vertex1]. Regions;   { The two region lists }
              Regions_Vertex2 := Graph^ [Vertex2]. Regions;
              Fragments [Loop]. Common_Count := 0;
              Fragments [Loop]. Common_Regions := Nil;            { no common regions so far }
              while (Regions_Vertex1 <> Nil)             { Search for the Common Regions }
                    and (Regions_Vertex2 <> Nil) do      { of the first two attachments  }
                begin
                  if Regions_Vertex1^. Region < Regions_Vertex2^. Region
                    then Regions_Vertex1 := Regions_Vertex1^. Next   { no match - move to next region }
                    else
                      if Regions_Vertex1^. Region = Regions_Vertex2^. Region { if the regions are equal }
                        then begin
                          if Fragments [Loop]. Common_Regions = Nil
                            then begin
                              new (Fragments [Loop]. Common_Regions);
                              Temp_Regions := Fragments [Loop]. Common_Regions;
```

```
                        end
                        else begin
                          new (Temp_Regions^. Next);
                          Temp_Regions := Temp_Regions^. Next
                        end;
                      Temp_Regions^. Next := Nil;                          { add the new common region }
                      Temp_Regions^. Region := Regions_Vertex1^. Region;
                      Fragments [Loop]. Common_Count := Fragments [Loop]. Common_Count + 1;   { increment
                      Regions_Vertex1 := Regions_Vertex1^. Next;           { and move to next region }
                      Regions_Vertex2 := Regions_Vertex2^. Next
                    end
                    else Regions_Vertex2 := Regions_Vertex2^. Next   { no match - move to next region }
              end;
            if Fragments [Loop]. Common_Count = 0      { check if the first two attachments have no regions
              then begin
                Planar := false
              end
              else Check_Rest_of_Attachments;   { now reduce the common list to represent }
          end;
        if Planar
          then if (Fragments [Loop]. Common_Count < Best_Count)   { see if we have a new best }
                 then begin
                   Best_Count := Fragments [Loop]. Common_Count;
                   Best_Fragment := Loop
                 end
      end
  end;
if Planar
  then begin                                { now set up the father pointers correctly }
    Current_Attach := Fragments [Best_Fragment]. Attachments;
    Graph^ [Current_Attach^. Vertex]. Father := 0;     { root has no father }
    Graph^ [Current_Attach^. Next^. Vertex]. Father := { Father of second attachment is saved }
             Current_Attach^. Next^. Father;
    Fragment_to_Use := Best_Fragment;         {-return the fragment to use }
    Region_to_Use :=
       Fragments [Best_Fragment]. Common_Regions^. Region; { and the region to embed the path into }
  end
end;
```

```
{-------------------------------------------------------}
{- The fragment to embed a path from, and the region to     -}
{- embed the path into was chosen in the above procedure.   -}
{- The path we choose from the fragment is very easy, it    -}
{- starts from the second attachment, and follows the father-}
{- pointers back up the fragment to the root.               -}
{- The main difficulty is the maintenance of the Region list-}
{- and vertex lists.                                        -}
{-------------------------------------------------------}
procedure Add_Path_to_Graph;

const Debug  = false;

var
  Made_Circle     : Boolean;          { if we have converted a linked list to circular list }
  Current_Attach,                     { All the rest are temporary variables }
  Current_Attach2,
  Temp_Attachment : Attach_List_Ptr;
  Temp_Edge       : Edge_Ptr;
  Temp_Tree,
  Vertex1,                            { First attachment }
  Vertex2         : Vertex_Ptr;       { Second attachment }
  Temp_Vertex,
  Temp_Vertex2,
  Start_Region1,                      { Start of the old region list }
  Start_Region2,                      { Start of the new Region list }
  End_Region1,                        { End of the old region list }
  End_Region2     : Vertex_List_Ptr;  { End of the new region list }

begin
  Vertex1 := Fragments [Fragment_To_Use]. Attachments^. Vertex;          { get the end vertex of the path }
  Vertex2 := Fragments [Fragment_to_Use]. Attachments^. Next^. Vertex;   { and the start vertex of the path
  Temp_Vertex := Region_Vertices [Region_to_Use];
  while (Temp_Vertex^. Vertex <> Vertex1) do   { search for vertex1 in the Region list }
    begin
      Graph^ [Temp_Vertex^. Vertex]. Mark := 1;       { Re-generate Common_Lists for this vertex }
      Temp_Vertex := Temp_Vertex^. Next;
    end;
  Graph^ [Temp_Vertex^. Vertex]. Mark := 3;
  Start_Region1 := Temp_Vertex;           { note the start of the first (old) Region }
  Made_Circle := false;                   { we have not converted the list to a circular list yet }
  while (Temp_Vertex^. Next^. Vertex <> Vertex2) do     { search until end vertex of Region1 }
    if Temp_Vertex^. Next = Nil           { if we reach the ned of the list first }
      then begin
        Made_Circle := true;
        Temp_Vertex^. Next := Region_Vertices [Region_to_Use]  { then make it a circular list }
      end
      else begin
        Graph^ [Temp_Vertex^. Vertex]. Mark := 2;     { Re-generate Common_Lists for this vertex }
        Temp_Vertex := Temp_Vertex^. Next
      end;
  Graph^ [Temp_Vertex^. Vertex]. Mark := 2;
  if not Made_Circle                { If we have not made the list to a circular }
    then begin                      { list yet then we complete the cycle.       }
      Temp_Vertex2 := Temp_Vertex^. Next;
      while Temp_Vertex2^. Next <> Nil do
        begin
          Graph^ [Temp_Vertex2^. Vertex]. Mark := 1;
```

```
        Temp_Vertex2 := Temp_Vertex2^. Next
      end;
    Graph^ [Temp_Vertex2^. Vertex]. Mark := 1;
    Temp_Vertex2^. Next := Region_Vertices [Region_to_Use]
  end;
Graph^ [Temp_Vertex^. Next^. Vertex]. Mark := 1;
                                    { At this point, all vertices that were bounding the }
                                    { Region Region_to_Use have been set so that the     }
                                    { fragment generation will include this vertex.      }
Start_Region2 := Temp_Vertex^. Next;
Region_Vertices [Region_to_Use] := Start_Region1;    { break links to form the old region's new list }
new (Temp_Vertex^. Next);
Temp_Vertex := Temp_Vertex^. Next;                   { Old Region starts at Vertex1 and goes to Vertex2 }
Temp_Vertex^. Next := Nil;
Temp_Vertex^. Vertex := Vertex2;                     { add last vertex in the list }
End_Region1 := Temp_Vertex;

Regions_Used := Regions_Used + 1;                    { Form the new Region }
Region_Vertices [Regions_Used] := Start_Region2;     { New region starts }
Temp_Vertex := Start_Region2;                        { at second vertex and goes to first vertex }
while (Temp_Vertex^. Next <> Start_Region1) do
  begin
    Add_Delete_to_Regions (Region_to_Use, Regions_Used,  { Delete the old Region   }
                           Temp_Vertex^. Vertex);        { and add the new Region. }
    Temp_Vertex := Temp_Vertex^. Next;
  end;

Add_Delete_to_Regions (Region_to_Use, Regions_Used,   { and change the last vertice's Region status }
                           Temp_Vertex^. Vertex);
new (Temp_Vertex^. Next);                             { replicate vertex1 for the region list }
Temp_Vertex := Temp_Vertex^. Next;
Temp_Vertex^. Next := Nil;
Temp_Vertex^. Vertex := Vertex1;
Add_Delete_to_Regions (Region_to_Use, Regions_Used,
                           Temp_Vertex^. Vertex);
End_Region2 := Temp_Vertex;                  { At this stage we have two linked lists }
                                             { The first represents the Old - from Vertex1 to Vertex2  }
                                             { The second represents the New - from Vertex2 to Vertex1 }
   { Now we add the vertices on the new path - from Vertex2 to Vertex1 }
Temp_Tree := Fragments [Fragment_to_Use]. Attachments^. Next^. Vertex;
while Graph^ [Temp_Tree]. Father <> 0 do      { While we have not hit the root yet }
  begin
    Temp_Edge := Graph^ [Temp_Tree]. Edges;
    while Temp_Edge^. Vertex <> Graph^ [Temp_Tree]. Father do   { Find edge corresponding to }
      Temp_Edge := Temp_Edge^. Next;                            { the edge in the fragment }
    Temp_Edge^. Used := true;
    Temp_Edge^. Other_Edge^. Used := true;
    Edges_Used := Edges_Used + 1;                          { add the edge to the graph H }
    if (Temp_Tree <> Vertex1) and (Temp_Tree <> Vertex2)  { check if the vertex is not an attachment }
      then begin
        Graph^ [Temp_Tree]. Used := true;
        Graph^ [Temp_Tree]. Gen_Frags := true;     { Mark the vertex added to the graph }
        Add_two_Regions_to_Vertex (Region_to_Use, Regions_Used,  { give it the two regions }
                                   Temp_Tree, false);           { but don't add to Region list }
        Number_in_H := Number_in_H + 1;
        Vertices_of_H [Number_in_H] := Temp_Tree;
        new (Temp_Vertex);                         { add the vertex to the region lists }
        Temp_Vertex^. Vertex := Temp_Tree;
```

```
          End_Region1^. Next := Temp_Vertex;          { for the old Region we add to the end of the list }
          Temp_Vertex^. Next := Nil;
          End_Region1 := Temp_Vertex;                   { move to the end }

          new (Temp_Vertex);
          Temp_Vertex^. Next := End_Region2^. Next;  { for the new region the path    }
          End_Region2^. Next := Temp_Vertex;          { must be added in reverse order.}
          Temp_Vertex^. Vertex := Temp_Tree;
        end
        else begin
          Add_Region_to_Vertex (Region_to_Use, Temp_Tree);  { with an attachment we add the new vertex only
          Graph^ [Temp_Tree]. Mark := 3;
          Fragments [Fragment_to_Use]. Attachments := Nil;
          Fragments [Fragment_to_Use]. Common_Regions := Nil;
          Fragments [Fragment_to_Use]. Common_Count := 0;
        end;
      Temp_Tree := Graph^ [Temp_Tree]. Father        { Backtrack up the fragment }
    end;
  Add_Region_to_Vertex (Region_to_Use, Temp_Tree);  { add the new Region to the Root = Vertex1 }
  Graph^ [Temp_Tree]. Mark := 3;
end;

{-------------------------------------------------------}
{- The fragments which are rooted at vertices which bounded -}
{- the Region in which the path was embedded (Region_to_Use)-}
{- are deleted, and set to nil.                          -}
{- This is because the fragments may no longer be valid with-}
{- the new path embedded.                                -}
{- In this case, we must note that Region_to_Use no longer  -}
{- valid.                                                -}
{-------------------------------------------------------}
procedure Update_Fragment_Status;

var
  Temp_Vertex       : Vertex_List_Ptr;
  Loop              : Vertex_Ptr;
  Temp_Region,
  Temp_Region2      : Region_List_Ptr;    { to update the region lists }
  Finished,
  R1, R2, Neutral,                        { the three classifications of fragments in this region }
  Delete_It         : Boolean;            { if we have to delete the fragment entirely }
  Current_Attach2,
  Current_Attach    : Attach_List_Ptr;
  Current_Frag_Posn : Integer;

begin
 Current_Frag_Posn := 1;
 while (Current_Frag_Posn <= Number_Frags_Found) do    { for every fragment do }
   begin                                                 { see if attachments over path we have added }
     Current_Attach := Fragments [Current_Frag_Posn]. Attachments;
     Delete_It := false;
     R1 := false; R2 := false; Neutral := false;       { assume in none of the new regions }
     Finished := (Current_Attach = Nil);
     while not Finished do                             { and find out }
       begin
         if Graph^ [Current_Attach^. Vertex]. Mark = 1   { Region 1 }
           then R1 := true
           else if Graph^ [Current_Attach^. Vertex]. Mark = 2  { Region 2 }
```

```
                then R2 := true
                else if Graph^ [Current_Attach^. Vertex]. Mark = 3  { end vertices of the path we added }
                       then Neutral := true
                       else Delete_It := true;       { harmless - leave alone }
        Current_Attach := Current_Attach^. Next;   { goto next attachment }
        Finished := (Current_Attach = Nil) or (Delete_It)
      end;
    if not Delete_It
      then
        if R1 and R2   { incompatible - so we must remove Region_to_use }
          then begin
            Delete_It := false;
            Temp_Region := Fragments [Current_Frag_Posn]. Common_Regions;
            if Temp_Region^. Region = Region_to_Use   { delete the region from list of regions }
              then begin
                Fragments [Current_Frag_Posn]. Common_Regions := Temp_Region^. Next;
                dispose (Temp_Region);
                Delete_It := true                     { note we have deleted }
              end
              else begin           { delete the region from the list of regions }
                while (Temp_Region^. Next <> Nil) and
                      ( Temp_Region^. Next^. Region <> Region_to_Use) do
                  Temp_Region := Temp_Region^. Next;
                if Temp_Region^. Next = nil
                  then Temp_Region := Nil
                  else begin
                    Delete_it := true;
                    Temp_Region2 := Temp_Region^. Next;
                    Temp_Region^. Next := Temp_Region2^. Next;
                    dispose (Temp_Region2)
                  end;
              end;
            if Delete_It        { must update the fragment's stats }
              then Fragments [Current_Frag_Posn]. Common_Count :=
                     Fragments [Current_Frag_Posn]. Common_Count - 1;
          end
          else if (R1)    { now in new region - remove old region and add new region }
                 then begin
                   Delete_It := false;
                   Temp_Region := Fragments [Current_Frag_Posn]. Common_Regions;
                   if Temp_Region^. Region = Region_to_Use        { delete old region }
                     then begin
                       Delete_It := true;
                       Fragments [Current_Frag_Posn]. Common_Regions :=
                                   Temp_Region^. Next;
                       Temp_Region2 := Temp_Region
                     end
                     else begin
                       while (Temp_Region^. Next <> Nil) and
                             (Temp_Region^. Next^. Region <> Region_to_Use) do
                         Temp_Region := Temp_Region^. Next;       { add new region on end - sorted }
                       if Temp_Region^. Next = nil
                         then
                         else begin
                           Delete_It := true;
                           Temp_Region2 := Temp_Region^. Next;
                           Temp_Region^. Next := Temp_Region2^. Next;
                         end;
```

```
                    end;
                  if Delete_It
                    then begin
                      if Fragments [Current_Frag_Posn]. Common_Regions = Nil
                        then Fragments [Current_Frag_Posn]. Common_Regions :=
                               Temp_Region2
                        else begin
                          while Temp_Region^. Next <> Nil do
                            Temp_Region := Temp_Region^. Next;
                          Temp_Region^. Next := Temp_Region2
                        end;
                      Temp_Region2^. Region := Regions_Used;
                      Temp_Region2^. Next := Nil;
                    end;
                end
                else if (Neutral) and (not R1) and (not R2)   { belongs to both regions }
                       then begin                             { so we add the new region }
                         Temp_Region := Fragments [Current_Frag_Posn]. Common_Regions;
                         while (Temp_Region <> Nil)
                           and (Temp_Region^. Region <> Region_to_Use) do
                           Temp_Region := Temp_Region^. Next;
                         if (Temp_Region <> Nil)
                           then begin
                             while Temp_Region^. Next <> Nil do
                               Temp_Region := Temp_Region^. Next;
                             new (Temp_Region^. Next);
                             Temp_Region^. Next^. Next := Nil;
                             Temp_Region^. Next^. Region := Regions_Used;
                             Fragments [Current_Frag_Posn]. Common_Count :=
                               Fragments [Current_Frag_Posn]. Common_Count + 1
                           end
                       end;
    Current_Frag_Posn := Current_Frag_Posn + 1     { try the next fragment }
  end;
  Temp_Vertex := Region_Vertices [Region_to_Use];   { lastly we must reset the values }
  while Temp_Vertex <> Nil do                       { of reduced old region }
    begin
      Graph^ [Temp_Vertex^. Vertex]. Mark := 0;
      Temp_Vertex := Temp_Vertex^. Next
    end;
  Temp_Vertex := Region_Vertices [Regions_Used];    { and of the new created region }
  while Temp_Vertex <> Nil do
    begin
      Graph^ [Temp_Vertex^. Vertex]. Mark := 0;
      Temp_Vertex := Temp_Vertex^. Next
    end
end;
```

```
{-------------------------------------------------------}
{- Main Demoucran, Malgrange and Pertuisel algorithm.  -}
{-------------------------------------------------------}
begin
  Initialise_DMP;            { Main Initialise }

  if Planar
    then begin
      if Current_Vertex < Last_Vertex
        then begin
          Find_a_Cycle;                                  { get the cycle, embed it }
          Finished := (Edges_Used = Max_Edges);
          while not Finished do
            begin
              writeln ('Currently done ', Edges_Used, ' out of ', Max_Edges, ' Edges.');
              Generate_Fragments;                        { Get the fragments left }
              Fragments_Generated := Check_Fragments_Left;   { If none are left, then finished }
              if Fragments_Generated
                then begin
                  Determine_Common_Regions;              { otherwise determine common regions }
                  if Planar
                    then begin
                      Add_Path_to_Graph;                 { embed the path chosen }
                      UpDate_Fragment_Status;            { and delete all affected fragments }
                      Finished := (Edges_Used = Max_Edges)
                    end
                    else Finished := true
                end
                else Finished := true              { if the fragments are finished then so are we }
            end;
        end;
      if not Planar
        then writeln ('The graph is non-planar')
        else writeln ('The graph is planar');
    end;
end;

end.
```

```
{-------------------------------------------------------------}
{- Main Hopcroft and Tarjan procedure                        -}
{-------------------------------------------------------------}
begin
  writeln ('Testing for Planarity');
  Initialise_Hopcroft_Tarjan;         { initialise }
  if Planar
    then begin
      PathFinder (Current);
      writeln;
      if not Planar
        then writeln ('The graph is non-planar')
        else writeln ('The graph is planar')
    end
end;

end.
```

```
{-------------------------------------------------------------}
{- This unit contains global routines used in the main program -}
{- and by the algorithms.                                      -}
{-------------------------------------------------------------}
unit Planar_Miscellaneous;

interface

procedure Prompt;             { waits for a keypress with a message }

procedure Dump_Graph;         { displays the current graph }

procedure Initialise_Graph;   { resets graph values to a empty graph }

procedure Build_Graph;        { reads in a graph from a disk file }

procedure Enter_Own_Graph;    { you enter a graph of your choice }

function Check_2_Connected : boolean; { Checks if graph is 2-connected }

procedure Generate_Random_Planar_Graph; { generates a random planar graph }

procedure Generate_Random_Non_Planar_Graph; { generates a random planar graph }


implementation

uses Planar_Defs,
     Dos,
     CRT;

type
  Face_Range = 1..2000;        { for random generation of graphs }

var
  Face      : Array [Face_Range] of record
                                    Vertex1, Vertex2, Vertex3 : Vertex_Ptr;
                                  end;
  Last_Face : Face_Range;

{-------------------------------------------------------------}
{- Prompts for a key                                          -}
{-------------------------------------------------------------}
procedure prompt;

var Dummy : char;

begin
  while keypressed do
    Dummy := Readkey;
  TextColor (Red);
  writeln ('Press any key to continue...');
  TextColor (Yellow);
  repeat until keypressed;
  Dummy := ReadKey;
end;
```

```
{-------------------------------------------------------}
{- The current graph is displayed on the screen        -}
{-------------------------------------------------------}
procedure Dump_Graph;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge   : Edge_Ptr;
  Scr         : integer;

begin
  clrscr;
  Scr := 0;
  writeln ('The Graph is ');
  writeln;
  writeln ('Physical  Logical   ST          Edges');
  For Temp_Vertex := 1 to Last_Vertex do  { for every vertex display the edges }
    begin
      write (Temp_Vertex:7, Graph^ [Temp_Vertex]. Number:9, '':4);
      write (Graph^ [Temp_Vertex]. ST_Number:3);
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      write (' ':8);
      while Temp_Edge <> Nil do
        begin
          write (Graph^ [Temp_Edge^. Vertex]. St_Number:4);
          Temp_Edge := Temp_Edge^. Next
        end;
      writeln;
      Scr := Scr + 1;
      if Scr = 20     { check for full screen and wait if necessary }
        then begin
          Prompt;
          Scr := 0;
        end;
    end;
  writeln ('and now back to fun! ....');
  writeln;
  prompt;
end;

{-------------------------------------------------------}
{- Set up an empty graph                               -}
{-------------------------------------------------------}
procedure Initialise_Graph;

var Loop : Vertex_Ptr;

begin
  new (Graph);
  For Loop := 1 to Max_Vertices do
    Graph^ [Loop]. Edges := Nil;
  Last_Vertex := 0;
end;
```

```
{-------------------------------------------------------}
{- A graph is read in from a file.                      -}
{-                                                      -}
{- The file structure is :                              -}
{-        Every line is of the form                     -}
{-                                                      -}
{- a b b b b b ....                                     -}
{-                                                      -}
{-        That is a single vertex number denoted 'a'    -}
{-              and vertices 'b' adjacent to 'a' follow.-}
{-        Note that if vertices u and v are adjacent, then -}
{-        you must only specify either the edge uv or vu, -}
{-        but not both. If both are specified, then two -}
{-        edges will be added to the graph.             -}
{-------------------------------------------------------}
procedure Build_Graph;

const
  StartRow   = 7;            { row to start display of options }

var
  MaxRow,
  MaxCol,
  Row, Col    : Byte;
  FileNames   : Array [1..10, 0..3] of string [12];
  DirInfo     : SearchRec;   { to search through the possible data files }
                             { temporary variables }
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;
  This_Vertex,
  This_Posn,
  Other_Posn,
  Other_Vertex: integer;
  FileName,
  Edge_Set    : string;      { set of edges adjacent from This_Vertex }
  Dummy       : char;
  Inp         : text;        { data file we are reading in from }

{-------------------------------------------------------}
{- This procedure allows the user to interactively enter the-}
{- graph name by panning around a screen of file names.  -}
{-------------------------------------------------------}
procedure Get_Filename;

var
  Current_Char : Char;
  Finished     : Boolean;

begin
  Gotoxy (1, StartRow + MaxRow + 1);
  TextColor (Red);
  writeln ('Use Arrow Keys to select a file, press <RETURN> or <Enter> to accept');
  TextColor (Yellow);
  Row := 1;
  Col := 0;
  Finished := false;
  repeat
    TextColor (LightRed);
```

```
    GotoXy (1 + 20 * Col, Row + StartRow - 1);    { highlight the current option }
    write (Filenames [Row, Col] : 16);
    Current_Char := Readkey;
    if Current_Char = #0
      then Current_Char := Readkey;
    TextColor (Yellow);
    GotoXy (1 + 20 * Col, Row + StartRow - 1);    { lowlight the current option }
    write (Filenames [Row, Col] : 16);
    TextColor (Black);
    case Current_Char of                          { get movement and move }
      #72 : begin  { up }
              if Row > 1
                then Row := Row - 1
                else Row := MaxRow
            end;
      #80 : begin  { down }
              if Row < MaxRow
                then Row := Row + 1
                else Row := 1
            end;
      #75 : begin  { left }
              if Col > 0
                then Col := Col - 1
                else if Row = MaxRow
                       then Col := MaxCol
                       else Col := 3
            end;
      #77 : begin  { right }
              if ((Row <> MaxRow) and (Col < 3)) or
                 ((Row = MaxRow) and (Col < MaxCol))
                then Col := Col + 1
                else Col := 0
            end;
      #79 : begin  { end }
              Col := MaxCol;
              Row := MaxRow
            end;
      #71 : begin  { home }
              Col := 0;
              Row := 1
            end;
      #13 : begin                                 { filename has been selected }
              Filename := Filenames [Row, Col];
              Finished := true
            end
      else begin                                  { error in the input }
        Sound (220);
        Delay (20);
        NoSound
      end
      end
  until Finished;
  GotoXY (1, StartRow + MaxRow + 6);
  TextColor (Yellow)
end;
```

```
begin
  clrscr;
  TextColor (Red);
  writeln;
  writeln;
  writeln ('                    The Available Files for Testing are :');
  TextColor (Yellow);
  writeln ('_____________________________________________________________________________');
  writeln;
  findfirst ('\Data\*.Dat', Archive, Dirinfo);
  Row := StartRow;
  Col := 0;
  while DosError = 0 do
    begin
      Filenames [Row - 6, Col] := DirInfo. Name;
      GotoXy (1 + Col*20, Row);
      write (DirInfo. Name : 16);
      if Col = 3
        then begin
          Col := 0;
          Row := Row + 1
        end
        else Col := Col + 1;
      FindNext (DirInfo)
    end;
  If Col = 0
    then begin
      MaxRow := Row - StartRow;
      MaxCol := 3
    end
    else begin
      MaxRow := Row - StartRow + 1;
      MaxCol := Col - 1
    end;
  GotoXy (1, MaxRow + StartRow);
  writeln;
  writeln;
  writeln ('_____________________________________________________________________________');
  writeln;
  Get_Filename;
  if Pos ('.', FileName) = 0
    then FileName := FileName + '.Dat';

  Global_Filename := Filename;
  FileName := '\Data\' + FileName;
  writeln ('Reading from ', FileName, '...');
  assign (inp, FileName);
  {$I-}
  Reset (Inp);
  {$I+}
  if (IOResult <> 0)
    then begin
      File_Loaded := false;
      Global_Filename := 'File Load Error';
      writeln ('File not found error.');
      prompt;
      exit
    end
```

```
    else begin
      writeln;
      writeln ('Reading ');
      File_Loaded := true;
      while not eof(inp) do       { for every vertex in the file }
        begin
          read (inp, This_Vertex);            { read in the vertex to start }
          write ('.');
          if Last_Vertex < This_Vertex        { keep track of graph order }
            then Last_Vertex := This_Vertex;
          while not eoln(inp) do              { for every vertex adjacent do }
            begin
              read (inp, Other_Vertex);
              new (Temp_Edge);
              Temp_Edge^. Used := false;      { add to edge list }
              Temp_Edge^. Deleted := false;
              Temp_Edge^. Next := Graph^ [This_Vertex]. Edges;
              Graph^ [This_Vertex]. Edges := Temp_Edge;
              Temp_Edge^. Vertex := Other_Vertex;

              if Last_Vertex < Other_Vertex
                then Last_Vertex := Other_Vertex;
              new (Temp_Edge2);               { and add edge to other vertex's edge list }
              Temp_Edge^. Other_Edge := Temp_Edge2;
              Temp_Edge2^. Other_Edge := Temp_Edge;
              Temp_Edge2^. Used := false;
              Temp_Edge2^. Deleted := false;
              Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
              Graph^ [Other_Vertex]. Edges := Temp_Edge2;
              Temp_Edge2^. Vertex := This_Vertex;
            end;
          readln (inp)
        end
    end;
  close (inp);              { exit neatly }
  writeln
end;
```

```
{-------------------------------------------------------------}
{- You enter a graph - procedure has extensive error checking-}
{-------------------------------------------------------------}
procedure Enter_Own_Graph;

var
  Finished_Line,
  Finished        : Boolean;
  Temp_Edge2,
  Temp_Edge    : Edge_Ptr;
  This_Vertex,
  Other_Vertex    : Vertex_Ptr;

{-------------------------------------------------------------}
{- Single charcter at a time parse to get an valid integer. -}
{-------------------------------------------------------------}
function Get_a_Number (var Fail_Status : Boolean) : Vertex_Ptr;

var
  Temp_Str    : String;       { string containing the number }
  Temp_Chr    : Char;         { a charcter of the string }
  Error,                      { error <> 0 if val fails }
  Temp_Int    : integer;      { the candidate integer }
  Got_Number : Boolean;       { success at getting candidate number }

begin
  Temp_Str := '';
  Got_Number := false;
  while (not eoln) and (not Got_Number) do    { get the number a character at a time }
    begin
      read (Temp_Chr);
      if Temp_Chr in ['0'..'9']
        then Temp_Str := Temp_Str + Temp_Chr
        else Got_Number := true
    end;
  if (Temp_Chr = ' ') or (Eoln)          { see if valid exit from get }
    then begin
      val (Temp_Str, Temp_Int, Error);
      Fail_Status := (Error <> 0)        { then see if valid number within range }
                  or (Temp_Int < 1)
                  or (Temp_Int > Max_Vertices);
      if not Fail_Status
        then Get_a_Number := Temp_Int;
    end
    else Fail_Status := true
end;
```

```
{-------------------------------------------------------}
{- Main Enter_own_Graph                                 -}
{-------------------------------------------------------}
begin
  Initialise_Graph;
  clrscr;
  TextColor (Red);
  writeln;
  writeln;
  writeln ('Please Enter the Graph by giving the adjacency lists of the vertices.');
  Writeln ('First enter the vertex that the adjacency list refers to. ');
  Writeln ('Then give all vertices adjacent to that vertex.');
  writeln;
  Writeln ('For example, if vertex 1 is adjacent to vertices 3, 4 and 5, then you enter');
  Writeln ('1 3 4 5');
  TextColor (Yellow);
  writeln ('_____________________________________________________________________________');
  writeln;

  writeln;
  File_Loaded := true;
  Finished := false;
  while not Finished do                    { for every vertex in the file }
    begin
      Writeln ('Enter a vertex number, followed by the adjacency list');
      Writeln ('or press ''Q'' to quit');
      Writeln ('In both cases please press <Enter> or <Return> when finished.');
      This_Vertex := Get_a_Number (Finished); { read in the vertex to start }
      if not Finished
        then begin
          if Last_Vertex < This_Vertex         { keep track of graph order }
            then Last_Vertex := This_Vertex;
          Finished_Line := false;

          repeat                               { for every vertex adjacent do }
            Other_Vertex := Get_a_Number (Finished_Line);
            if not Finished_Line
              then begin
                Temp_Edge := Graph^ [This_Vertex]. Edges;    { first check if edge already there }
                while (Temp_Edge <> nil) and (Temp_Edge^. Vertex <> Other_Vertex) do
                  Temp_Edge := Temp_Edge^. Next;
                if (Temp_Edge = nil) and (Other_Vertex <> This_Vertex) { if not duplicate edge then add it }
                  then begin
                    new (Temp_Edge);
                    Temp_Edge^. Used := false;       { add to edge list }
                    Temp_Edge^. Deleted := false;
                    Temp_Edge^. Next := Graph^ [This_Vertex]. Edges;
                    Graph^ [This_Vertex]. Edges := Temp_Edge;
                    Temp_Edge^. Vertex := Other_Vertex;

                    if Last_Vertex < Other_Vertex
                      then Last_Vertex := Other_Vertex;
                    new (Temp_Edge2);                { and add edge to other vertex's edge list }
                    Temp_Edge^. Other_Edge := Temp_Edge2;
                    Temp_Edge2^. Other_Edge := Temp_Edge;
                    Temp_Edge2^. Used := false;
                    Temp_Edge2^. Deleted := false;
                    Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
```

```
                    Graph^ [Other_Vertex]. Edges := Temp_Edge2;
                    Temp_Edge2^. Vertex := This_Vertex
                  end
                  else if Other_Vertex = This_Vertex
                        then writeln ('Sorry - multigraphs are not accepted.');
              end
          until Finished_Line;    { until read entire adjacency list in }
          readln
        end
    end;
  writeln;
  writeln ('Finished Reading in the Graph');
  writeln;
  if Last_Vertex = 0
    then File_Loaded := false;
  prompt
end;

{-------------------------------------------------------------}
{- The Graph checked for 2-connectivity.                      -}
{-                                                            -}
{- We look at the LowPoint number L1 of each vertex generated -}
{- by the Depth First Search. If L1 = the DFS number of the   -}
{- vertex, then obviously there is no path from this vertex   -}
{- to further up the tree. i.e. the vertex is a cut vertex.   -}
{-                                                            -}
{- See Chapter 1 for more details.                            -}
{-------------------------------------------------------------}
Function Check_2_Connected : Boolean;

var
  Count_Kids : integer;
  Loop       : Vertex_Ptr;
  Temp_Edge,
  Temp_Edge2 : Edge_Ptr;

begin
  ClrScr;
  writeln ('Checking that the Graph is 2-connected.');
  if Last_Vertex > 3
    then
      For Loop := 1 to Last_Vertex do
        with Graph^ [Loop] do
          begin
            if (L1 = Number) and (Number <> 1)   { check Low Point value of not the root }
              then begin
                writeln ('This graph is not 2-connected');
                writeln ('A further DFS is necessary to separate into 2-connected components');
                Check_2_Connected := false;
                exit
              end
              else if Number = 1                 { check if root has more than one child }
                    then begin
                      Temp_Edge := Edges;
                      Count_Kids := 0;
                      while Temp_Edge <> nil do
                        begin
                          if not Temp_Edge^. Deleted
```

```
                        then Count_Kids := Count_Kids + 1;
                      Temp_Edge := Temp_Edge^. Next
                    end;
                  if Count_Kids > 1
                    then begin
                      writeln ('This graph is not 2-connected');
                      writeln ('A further DFS is necessary to separate into 2-connected components');
                      Check_2_Connected := false;
                      exit
                    end
                end
          end
      else begin
        Check_2_Connected := false;
        writeln ('This graph is not 2-connected');
        writeln ('A further DFS is necessary to separate into 2-connected components');
        exit
      end;
  Check_2_Connected := true
end;

{-------------------------------------------------------------}
{- An edge is added between vertex1 and vertex2              -}
{-------------------------------------------------------------}
procedure Add_Edge (Vertex1, Vertex2 : Vertex_Ptr);

var
  Temp_Edge2,
  Temp_Edge : Edge_Ptr;

begin
  new (Temp_Edge);
  new (Temp_Edge2);
  with Temp_Edge^ do
    begin
      Vertex := Vertex2;
      Next := Graph^ [Vertex1]. Edges;
      Other_Edge := Temp_Edge2;
      Deleted := false;
      Used := false;
    end;
  Graph^ [Vertex1]. Edges := Temp_Edge;
  with Temp_Edge2^ do
    begin
      Vertex := Vertex1;
      Next := Graph^ [Vertex2]. Edges;
      Other_Edge := Temp_Edge;
      Deleted := false;
      Used := false;
    end;
  Graph^ [Vertex2]. Edges := Temp_Edge2;
end;
```

```
{-------------------------------------------------------}
{- A random large graph is generated.                  -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{-------------------------------------------------------}
procedure Generate_Random_Graph (var Last_Face : Face_Range;
                                 Vertices_Placed : Integer);

var
  Loop      : Integer;
  Temp_Edge : Edge_Ptr;
  Current   : Vertex_Ptr;
  Current_Face : 1..2000;

begin
  Current := Vertices_Placed;
  while Current < Last_Vertex do     { for each new vertex do }
    begin
      Current := Current + 1;
      Current_Face := Random (Last_Face) + 1;      { choose a random face }
      with Face [Current_Face] do
        begin
          Graph^ [Current]. Edges := Nil;
          Add_Edge (Vertex1, Current);             { add edges from new vertex to }
          Add_Edge (Vertex2, Current);             { the three vertices on the face }
          Add_Edge (Vertex3, Current);
          Last_Face := Last_Face + 1;
          Face [Last_Face]. Vertex1 := Vertex2;    { we now have two new faces }
          Face [Last_Face]. Vertex2 := Vertex3;
          Face [Last_Face]. Vertex3 := Current;
          Last_Face := Last_Face + 1;
          Face [Last_Face]. Vertex1 := Vertex1;
          Face [Last_Face]. Vertex2 := Current;
          Face [Last_Face]. Vertex3 := Vertex3;
          Vertex3 := Current;
        end;
    end;
  Last_Vertex := Current;
end;
```

```
{-------------------------------------------------------}
{- A random large maximal planar graph is generated.   -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{- Our initial graph is a triangle.                    -}
{-------------------------------------------------------}
procedure Generate_Random_Planar_Graph;

var
  loop : integer;

begin
  Randomize;
  File_Loaded := true;
  Initialise_Graph;
  clrscr;
  writeln ('Please enter the size of the Graph to create');
  readln (Last_Vertex);        { get order of graph }
  writeln;
  Graph^ [1]. Edges := Nil;
  Graph^ [2]. Edges := Nil;
  Graph^ [3]. Edges := Nil;
  Add_Edge (1, 2);             { form the triangle }
  Add_Edge (1, 3);
  Add_Edge (2, 3);
  Face [1]. Vertex1 := 1;      { note the vertices on the face }
  Face [1]. Vertex2 := 2;
  Face [1]. Vertex3 := 3;
  Last_Face := 1;
  Generate_Random_Graph (Last_Face, 3);
end;
```

```
{-------------------------------------------------------}
{- A random large non-planar graph is generated.       -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{- We start with a graph K5 minus one edge. We then generate-}
{- the maximal planar graph using the same method as above. -}
{- Lastly, we add the extra edge.                      -}
{-------------------------------------------------------}
procedure Generate_Random_Non_Planar_Graph;

begin
  Randomize;
  File_Loaded := true;
  Initialise_Graph;
  clrscr;
  writeln ('Please enter the size of the Graph to create');
  readln (Last_Vertex);        { get order of graph }
  writeln;
  Graph^ [1]. Edges := Nil;
  Graph^ [2]. Edges := Nil;
  Graph^ [3]. Edges := Nil;
  Graph^ [4]. Edges := Nil;
  Graph^ [5]. Edges := Nil;
  Add_Edge (1, 2);              { we now add all the edges of K5 to the graph }
  Add_Edge (1, 3);
  Add_Edge (1, 4);
  Add_Edge (1, 5);
  Add_Edge (2, 3);
  Add_Edge (2, 4);              { this edge is no added to the face tables below }
  Add_Edge (2, 5);
  Add_Edge (3, 4);
  Add_Edge (3, 5);
  Add_Edge (4, 5);
  Face [1]. Vertex1 := 1;       { Now we note the three vertices bounding each }
  Face [1]. Vertex2 := 2;       { face. We Emphasise that the Edge 2,4 is NOT  }
  Face [1]. Vertex3 := 5;       { added into the face tables. i.e. the face    }
  Face [2]. Vertex1 := 1;       { information relates to K5 - edge (2,4).      }
  Face [2]. Vertex2 := 5;
  Face [2]. Vertex3 := 4;
  Face [3]. Vertex1 := 3;
  Face [3]. Vertex2 := 4;
  Face [3]. Vertex3 := 5;
  Face [4]. Vertex1 := 2;
  Face [4]. Vertex2 := 3;
  Face [4]. Vertex3 := 5;
  Face [5]. Vertex1 := 1;
  Face [5]. Vertex2 := 3;
  Face [5]. Vertex3 := 4;
  Face [6]. Vertex1 := 1;
  Face [6]. Vertex2 := 2;
  Face [6]. Vertex3 := 3;
  Last_Face := 6;
  Generate_Random_Graph (Last_Face, 5);  { Now generate the appropriate random maximal graph }
end;
```

```
end.
```

```
{-------------------------------------------------------------------}
{- This unit contains global definitions used by all the algorithms -}
{-------------------------------------------------------------------}
unit Planar_Defs;

interface

const
  Max_Vertices = 200;    { maximum number of vertices on the graph }

type
  Vertex_Ptr        = 1..Max_Vertices;          { used to access a vertex }
  Vertex_Ptr_Range  = 0..Max_Vertices;

  {*************************************************************************}
                                    { The following are used in the      }
                                    { Demoucran, Malgrange and Pertuisel }
                                    { algorithm only                     }
  Region_Range     = 0..Max_Vertices;
  Region_List_Ptr  = ^Region_List;       { A linked list of regions is used }
  Region_List      = record              { for each vertex                  }
                       Next   : Region_List_Ptr;
                       Region : Region_Range;
                     end;
  Vertex_List_Ptr  = ^Vertex_List;       { A linked list of vertices is used }
  Vertex_List      = record              { for each region                   }
                       Next   : Vertex_List_Ptr;
                       Vertex : Vertex_Ptr;
                     end;

  Attach_List_Ptr  = ^Attach_List;       { for every fragment we store the }
  Attach_List      = record              { list of attachments and their parents }
                       Next   : Attach_List_Ptr;
                       Father : Vertex_Ptr_Range;
                       Vertex : Vertex_Ptr
                     end;
  Attachments      = array [Vertex_Ptr] of Record        { The following information is used for fragments }
                       Attachments    : Attach_List_Ptr; { The attachment list for each fragment }
                       Common_Count   : Integer;         { A count of the number of common regions }
                       Common_Regions : Region_List_Ptr  { And a list of the common regions }
                     end;


  PQ_Node_Ptr      = ^PQ_Node;           { used in Lempel, Even and Cederbaum }

  {*************************************************************************}
                                    {The General graph structure is next }
  Edge_ptr    = ^edge;              { for each vertex we store a list of edges }
  Edge        = record
                  Virtual_Number : integer;
                  Mark,             { Used in all algorithms for various tasks }
                  Weight : Vertex_Ptr_Range;  { Weighting for edge sortings     }
                  Deleted,          { we delete edges in Hopcroft, Tarjan      }
                                    { in the DFS routine                       }
                  Frag_Used,
                  Used   : boolean; { Boolean variable to note if the edge     }
                                    { has been tested in the planarity process }
                  Vertex : Vertex_Ptr;    { the vertex this edge is incident to      }
```

```
                PQ_Ptr : PQ_Node_Ptr;   { in Demoucran et al. - a pointer to the   }
                                        { PQ-tree leaf represented by this edge    }
                Other_Edge,             { Points to the edge element in the other  }
                                        { vertex's edge list                       }
                Next   : Edge_ptr;      { Next edge element in the edge list       }
              end;

  vertex     = record                   { each vertex has the following information }
                Edges   : edge_ptr;     { the edge list of edges incident to vertex }
                Regions : Region_List_Ptr; { regions this vertex is on }
                Mark,                   { Used in all algorithms in various tasks   }
                Mark1   : integer;
                Father  : Vertex_Ptr_Range; { The Father in the graph for DFS or in }
                                        { Demoucran et al. for fragment generating  }
                L1,                     { The weighting L1 used in                  }
                                        { Hopcroft & Tarjan and Lempel et al        }
                L2,                     { The second lowpoint used only in Hopcroft }
                ST_Number,              { Used in Lempel, Even and Cederbaum only   }
                Number  : Vertex_Ptr_Range; { depth first number                    }
                Gen_Frags,              { to generate fragments from this vertex }
                Used    : Boolean;      { if the vertex is used in the current graph}
              end;
  Graph_Ptr  = ^Graph_Structure;
  Graph_Structure = array [Vertex_Ptr] of Vertex;  { the graph is placed on the heap }

  {*************************************************************************}
                                        { The buckets are used in the        }
                                        { Hopcroft et al and Lempel et al    }
                                        { to perform a linear sort of edges  }
  Bucket_Ptr = ^Bucket;
  Bucket     = record
                Next   : Bucket_Ptr;    { the next edge with the same weighting }
                Vertex : Vertex_Ptr;    { the vertex this edge comes from       }
                Data   : Edge_Ptr;      { and the edge element itself           }
              end;

  {*************************************************************************}
                                        { The Stacks are used in Hopcroft    }
  Stack_Ptr  = ^Stack;
  Stack      = record                   { This stack has the fragments on it }
                Lowest_Path_Entry : Vertex_Ptr; { The lowest attachment      }
                Next              : Stack_Ptr;  { next fragment on this stack}
              end;
  Block_Stack_Ptr = ^Block_Stack;
  Block_Stack     = record              { This stack has blocks of fragments }
                      Lowest_Pi1,                  { the lowest on inner and }
                      Lowest_Pi2 : Stack_Ptr;      { outer stacks            }
                      Next       : Block_Stack_Ptr; { Next block of fragments }
                    end;

  {*************************************************************************}
                                        { this section describes the PQ tree }
                                        { data structures used in the        }
                                        { Lempel, Even and Cederbaum alg.    }

  PQ_Type    = (P_Node, Q_Node, Leaf);         { Node can be P-node, Q-node, leaf }
  Pass_Two_Status = (Empty, Full, Partial);              { all possible states }
  Pass_One_Status = (None, Queued, Blocked, UnBlocked); { for the two passes  }
```

```
  Double_Ptr      = ^Double_Ptr_Node;
  Double_Ptr_Node = record                { a circular double linked list for  }
                                          { a P-nodes children                 }
                      Left,                  { elements to the left and right  }
                      Right   : Double_Ptr; { of the node in the circular list}
                      Element : PQ_Node_Ptr { and the particular PQ-node child}
                    end;
  List_Ptr        = ^List;                { a singly linked list }
  List            = record
                      Next    : List_Ptr;
                      Element : PQ_Node_Ptr;
                    end;
  PQ_Node     = record
                  Data_Label         : Pass_Two_Status;
                  Full_Kids          : List_Ptr;    { a list of the full children  }
                  Full_Kids_Count    : Integer;     { a count of the full children }
                  Partial_Kids       : List_Ptr;    { partial kids - 0, 1, 2 nodes }
                                                    { never more than 2 nodes      }
                  Immediate_Siblings : List_Ptr;    { siblings - 0, 1, 2 nodes     }
                                                    { zero if the father is P-node }
                                                    { 1 if it is an endmost kid    }
                                                    { 2 otherwise                  }
                  Mark               : Pass_One_Status;
                  Parent             : PQ_Node_Ptr; { parent in the tree           }
                  Pert_Child_Count,                 { number of children pertinent }
                  Pert_Leaf_Count    : Integer;     { number descendant full leaves}
                  Circ_List_Posn     : Double_Ptr;  { points to a P-node parent    }
                  case Node_Type : PQ_Type of       { double circular list posn    }
                    P_Node    : (Child_Count   : Integer;       { number of children  }
                                 List_Start    : Double_Ptr);   { and circular list   }
                                                                { of the children     }
                    Q_Node    : (LeftMost_Kid  : PQ_Node_Ptr;   { two endmost children}
                                 RightMost_Kid : PQ_Node_Ptr);
                    Leaf      : (Tail_Vertex   : Vertex_Ptr);
                  end;


var
  Global_Filename : string;          { name of test file               }
  File_Loaded     : Boolean;         { if a graph is in memory         }
  Graph           : Graph_Ptr;       { the actual graph                }
  Last_Vertex     : Vertex_ptr_Range;{ the order of the current graph   }


implementation

begin
end.
```

```
program Convex_Drawing_of_Graphs;

{-------------------------------------------------------}
{-            The Driver Program                       -}
{-------------------------------------------------------}

uses
  Planar_Defs,          { Graph and type definitions }
  Planar_Miscellaneous, { Global Routines used by all the algorithms }
                        { The Algorithm units :- }
  Convex_Testing,
  CRT;                  { System unit for Screen control }


procedure Say_Bye;

begin
  Clrscr;
  GotoXY (30, 10);
  Writeln ('Returning to the Batch File');
  TextColor (Black); Write (' ');
  Delay (300)
end;

{-------------------------------------------------------}
{- The procedure is called during the Demonstration mode.   -}
{- We present a brief introductory screen giving a summary  -}
{- explanation of how the Demonstration is supposed to work.-}
{-------------------------------------------------------}
procedure Demo_Menu;

{-------------------------------------------------------}
{- We Print the actual information to do with the Demo.     -}
{-------------------------------------------------------}
procedure PrintInfo;

begin
  TextBackground(Black);
  Clrscr;
  TextColor(Red);
  Writeln ('Welcome to the second Project Demonstration Program');
  Writeln;
  Writeln ('                The Drawing Program');
  Writeln;
  Writeln;
  Writeln ('The Demonstration consists of a series of test files which');
  Writeln ('               will be drawn on the screen.');
  Writeln;
  Writeln ('The demonstration will pause after each stage to allow you to ');
  Writeln (' study the output. Please press any key to continue when finished.');
  Writeln;
  Writeln;
  Writeln;
  Writeln;
  Prompt
end;
```

```
begin
  FileNumber := 1;      { The first demo graph we shall use }
  PrintInfo;            { display intro page }
  repeat
    TextBackground (Black);
    ClrScr;
    TextColor (Yellow);
    Build_Graph;        { read in graph number Filenumber }
    if File_Loaded      { if successful ---> which it should always be }
      then begin
        mark (Heap_Position);      { note current memory allocation status }
        Convex_Test_Graph;         { test / find a convex cycle and hence draw graph }
        release (Heap_Position);   { release memory allocated during call }
        Scratch_Graph;             { dispose old graph }
        Initialise_Graph           { re-initialise for a new graph }
      end
      else begin
        writeln;
        writeln ('Fatal Error in the Demonstration Program');
        writeln ('This should not happen!!!!');
        writeln;
        writeln ('Please check the integrity of the Demonstration Floppy');
        writeln ('    or try to run the demonstration on another machine');
        prompt;
      end;
  until FileNumber > MaxDemoFiles; { until we have finished with all the demos }
  Clrscr;
  TextColor(Red);
  (*
  Writeln ('Press ''Y'' to enter your own graph, or press any key to continue');
  Writeln;
  if Upcase(Readkey) = 'Y'
    then begin
      mark (Heap_Position);      { note current memory allocation status }
      FileNumber := FileNumber + 1;
      Enter_Own_Graph;
      if (File_Loaded)
        then Convex_Test_Graph  { test / find a convex cycle and hence draw graph }
        else;
      release (Heap_Position)   { release memory allocated during call }
    end
  *)
end;
```

```
{-----------------------------------------------------------}
{- Displays the menu and accepts user's choice.            -}
{-----------------------------------------------------------}
procedure Menu;

var
  ReDraw,
  Finished : Boolean;
  Choice   : Char;

begin
  Finished := False;
  repeat
    TextBackground (Blue);
    ClrScr;
    Textcolor (Red);
    writeln ('                      Graph Drawing Algorithms');
    writeln ('                                                                   M.Sc. Project II');
    writeln ('ááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááá');
    writeln;
    write ('Current File : ' : 75 - Length (Global_Filename));
    TextColor (Yellow);
    writeln (Global_Filename);
    writeln;
    write ('':8);
    writeln ('1. S)how Current Graph');
    write ('':8);
    writeln ('2. R)ead in a new Graph');
    write ('':8);
    writeln ('3. P)lanar Graph generation');
    write ('':8);
    writeln ('4. N)on-Planar Graph generation');
    write ('':8);
    writeln ('7. T)est graph for convex extendability');
    writeln;
    write ('':8);
    writeln ('9. Q)uit');
    TextColor (Red);
    write (#10#10#10#10#10,'           Please enter your Choice [ ]', #8,#8);
    TextColor (Yellow);
    repeat
      ReDraw := true;
      Choice := UpCase (ReadKey);
      if Choice >= #32
        then write (Choice, #8)
        else write (' ', #8);
      case Choice of
        '1', 'S'      : if File_Loaded
                          then Dump_Graph      { displays graph structure }
                          else begin
                            writeln;
                            writeln (Global_Filename);
                            prompt;
                          end;
        '2', 'R'      : begin
                          Scratch_Graph;    { wipe old graph }
                          Initialise_Graph; { reset values }
                          Build_Graph;      { and read in new graph }
```

```
                    end;
       '3', 'P'     : begin
                      Scratch_Graph;
                      Initialise_Graph;
                      Generate_Random_Planar_Graph { generate a random planar }
                    end;
       '4', 'N'     : begin
                      Scratch_Graph;
                      Initialise_Graph;
                      Generate_Random_Non_Planar_Graph { generate a random non-planar }
                    end;
       '7', 'T'     : if File_Loaded
                      then begin
                        mark (Heap_Position);
                        Convex_Test_Graph;
                        release (Heap_Position);
                        Scratch_Graph;
                        Initialise_Graph
                      end
                      else begin
                        writeln;
                        writeln ('No File Loaded');
                        prompt;
                      end;
       '9', #27, 'Q' : Finished := true;
       else  Redraw := False
     end;
   until (ReDraw) or (Finished)
 until Finished;
end;

{-------------------------------------------------------------}
{- The Main Program                                          -}
{-------------------------------------------------------------}
begin
  Global_Filename := 'No File Loaded';    { for the interactive mode }
  File_Loaded := false;
  Initialise_Graph;                       { Reset original graph values }
  Do_Demo := true;
  if Do_Demo                              { and call relevant menu }
    then begin
      Demo_Menu;
      Say_Bye
    end
    else Menu
end.
```

```
{-------------------------------------------------------}
{- This unit                                           -}
{-      finds an embedding of the graph                -}
{-      tests if it has a convex cycle                 -}
{-      find the convex cycle                          -}
{-  and finally draws the graph in a convex manner.     -}
{-------------------------------------------------------}
unit Convex_Testing;


interface

procedure Convex_Test_Graph;


implementation

uses
  CRT,                  { standard TURBOPascal unit }
  Planar_Defs,          { all type definitions and global variables }
  Planar_Miscellaneous, { all miscellaneous services routines }
  Component_Handling,   { handling of split components and triconnected components }
  Hop_Algorithm,        { Finds split components and separation pairs }
  Embedd_Algorithm,     { finds an embedding of a graph and tests for planarity }
  Convex_Globals,       { global service routines to the convex testing and drawing units }
  Convex_Draw;          { Unit to draw the graph convex given an extendible cycle }


type
  Candidate_Range          = 1..255;                  { the name of the candidate triconnected component }
  Separation_Pair_List_Ptr = ^Separation_Pair_List;
  Separation_Pair_List     = record
                               Head, Tail    : Vertex_Ptr;      { head and tail of the separation pair }
                               Counts        : Array [Component_Types] of byte; { counts types of components
                               Other_Count   : 0..2;                            { number of 3-con. comps }
                               Other_Sources : Array [1..2] of Candidate_Range; { name specific 3-con. comps
                                                                                { used in cycle finding }
                               Ring_Source   : Candidate_Range;                 { name ring component }
                               Next          : Separation_Pair_List_Ptr
                             end;

var
  Pair_List        : Separation_Pair_List_Ptr;     { list of separation pairs }
  Extendible_Cycle : Boolean;                      { is there an extendible cycle }
  Number_Critical  : integer;                      { the number of critical pairs }


{-------------------------------------------------------}
{- General initialise                                  -}
{-------------------------------------------------------}
procedure Initialise;

{-------------------------------------------------------}
{- This procedure is called only if we are in the demo mode -}
{- The procedure prints the specific information about each -}
{- graph that we are testing.                          -}
{-------------------------------------------------------}
procedure Print_Info_File (The_Graph : integer);
```

```
begin
  TextColor(Red);
  writeln ('Information about this test file :');
  writeln;
  TextColor(Yellow);
  case The_Graph of
    1 : begin
          writeln ('Graph ', The_Graph, ' is a subgraph of the Kuratowski graph K');
          writeln ('      ', ' '      , '                                          5');
          writeln;
          writeln (' The graph has the property that all cycles are extendible');
          writeln
        end;
    2 : begin
          writeln ('Graph ', The_Graph, ' is a subgraph of the Kuratowski graph K');
          writeln ('      ', ' '      , '                                        3,3');
          writeln;
          writeln (' The graph has the property that all cycles are extendible');
          writeln
        end;
    3 : begin
          writeln ('Graph ', The_Graph, ' has 9 vertices');
          writeln;
          writeln (' The graph has the property that all cycles are extendible');
          writeln (' and we have to temporarily remove two vertices of degree 2 ');
          writeln (' during the algorithm.');
          writeln
        end;
    4 : begin
          writeln ('Graph ', The_Graph, ' has 15 vertices');
          writeln;
          writeln (' The graph has the property that all cycles are extendible');
          writeln (' and we have to temporarily remove two vertices of degree 2 ');
          writeln (' during the algorithm.');
          writeln
        end;
    5 : begin
          writeln ('Graph ', The_Graph, ' has 13 vertices');
          writeln;
          writeln (' The graph has the property that only a special cycle is extendible');
          writeln (' we must find a single cycle that has all separation pairs on it.');
          writeln
        end;
    6 : begin
          writeln ('Graph ', The_Graph, ' has 13 vertices');
          writeln;
          writeln (' The graph does not have an extendible convex cycle');
          writeln
        end;
    7 : begin
          writeln ('Graph ', The_Graph, ' has 25 vertices');
          writeln;
          writeln (' The graph has the property that only a special cycle is extendible');
          writeln (' we must find a single cycle that has the single separation pair on it,');
          writeln (' and we have to remove two vertices of degree 2 during the algorithm.');
          writeln
        end;
```

```
   8 : begin
         writeln ('Graph ', The_Graph, ' has 27 vertices');
         writeln;
         writeln (' The graph has the property that only a special cycle is extendible');
         writeln (' we must find a single cycle that has the single separation pair on it,');
         writeln (' and we have to remove two vertices of degree 2 during the algorithm.');
         writeln
       end
  end;
  writeln;
  writeln ('ááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááááá');
  writeln;
  prompt;
end;

var
  Temp_Edge   : Edge_Ptr;
  Temp_Vertex : Vertex_Ptr;

{-------------------------------------------------------}
{- Begin of Initialise Routine                         -}
{-------------------------------------------------------}
begin
  Clrscr;
  Pair_List := nil;          { no separation pairs yet }
  for Temp_Vertex := 1 to Last_Vertex do
    begin                    { reset some temporary variables }

      Graph^ [Temp_Vertex]. Deleted := false;
      Graph^ [Temp_Vertex]. Critical := false;  { no critical pairs yet }
      Graph^ [Temp_Vertex]. Component_ID := 0;   { no split components yet }
      Graph^ [Temp_Vertex]. Placed := false;
      Graph^ [Temp_Vertex]. Queued := false;
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while Temp_Edge <> nil do
        begin
          Temp_Edge^. Drawn := false;      { no edges drawn yet! }
          Temp_Edge := Temp_Edge^. Next
        end
    end;
  Forbidden_Pair := false;   { no forbidden pairs yet }
  Number_Critical := 0;      { 0 critical }
  Num_Deleted := 0;          { no vertces temporarily deleted }
  Extendible_Cycle := true;  { assume we can find a cycle }
  Is_2_Connected := false;
  if (Do_Demo) and (FileNumber <= MaxDemoFiles+1)
    then Print_Info_File (FileNumber-1)
end;

{-------------------------------------------------------}
{- Call the relevant routine to determine separation pairs  -}
{-------------------------------------------------------}
procedure Find_Separation_Pairs;

begin
  Hopcroft_Tarjan_Split_Components;     { we call the Hopcroft and Tarjan split components routine }
end;
```

```
{-------------------------------------------------------}
{- Scan through each split component determining its type   -}
{- i.e. ring, bond or 3-connected (which we cal other).     -}
{- Then analyse the results to determine critical pairs.    -}
{-------------------------------------------------------}
procedure Determine_Critical_Pairs;

var
  Head,
  Tail       : Vertex_Ptr;                  { head and tail of separation pair }
  Temp_Edge  : Component_Edge_Ptr;
  Base_Temp  : Separation_Pair_List_Ptr;    { temporary vars for separation pair lists }
  Candidate  : integer;                     { the candidate triconnected component }
  Finished   : Boolean;                     { finished checking all pairs }

{-------------------------------------------------------}
{- Swop two numbers                                     -}
{-------------------------------------------------------}
procedure Swop (var a, b : Vertex_Ptr);

var
  Temp : Vertex_Ptr;

begin
  Temp := a;
  a := b;
  b := Temp
end;

{-------------------------------------------------------}
{- A ring which has more than one virtual edge is not a ring-}
{- in the Chiba, Nishizeki (x, y) - split component sense.  -}
{- Here we check for this case, and modify the split comp.  -}
{- markers accordingly.                                     -}
{-------------------------------------------------------}
procedure Check_Triangles;

var
  Candidate,
  Virtual_Count : byte;
  Temp_Edge     : Component_Edge_Ptr;

begin
  For Candidate := 1 to Max_Candidates do               { for every triconnected component }
    if Components [Candidate]. Comp_Label = Triangle    { only necessary to check triangles }
      then begin
        Temp_Edge := Components [Candidate]. Edges;     { check edges for virtual or not }
        Virtual_Count := 0;
        repeat                                          { check every edge of ring }
          if Temp_Edge^. Origin <> Ordinary             { note each virtual edge }
            then Virtual_Count := Virtual_Count + 1;
          Temp_Edge := Temp_Edge^. Next
        until (Temp_Edge = nil) or (Virtual_Count > 1);
        if Virtual_Count > 1                              { relabel the component }
          then Components [Candidate]. Comp_Label := Other  { if necessary }
      end
end;
```

```
{-------------------------------------------------------}
{- Virt_Head > Virt_Tail is a separation pair. Add to the   -}
{- list of separation pairs if necessary. Update the stats  -}
{- of the component types for that particular pair.         -}
{-------------------------------------------------------}
procedure Adjust_Pair_List (Virt_Head, Virt_Tail : Vertex_Ptr);

var
  Temp_List,
  Temp_List2 : Separation_Pair_List_Ptr;

begin
  Base_Temp^. Next := Pair_List;            { dummy start of list to facilitate search }
  Temp_List := Base_Temp;
  while (Temp_List^. Next <> Nil) and       { scan thru list until correct place }
        ((Temp_List^. Next^. Head < Virt_Head) or
         ((Temp_List^. Next^. Head = Virt_Head) and
         (Temp_List^. Next^. Tail < Virt_Tail))) do
    Temp_List := Temp_List^. Next;
  if (Temp_List^. Next = Nil) or              { check if element is not }
     (Temp_List^. Next^. Head <> Virt_Head) or { already entered in list }
     (Temp_List^. Next^. Tail <> Virt_Tail)
    then begin
      new (Temp_List2);           { make new entry }
      with Temp_List2^ do
        begin
          Head := Virt_Head;      { note head }
          Tail := Virt_Tail;      { and tail  }
          Next := Temp_List^. Next;  { and insert }
          FillChar (Counts, Sizeof(Counts), 0);
          Other_Count := 0;
          Counts [Components [Candidate]. Comp_Label] := 1;   { update count for component }
          if (Components [Candidate]. Comp_Label = Triangle)
            then Ring_Source := Candidate                     { note origin of ring }
            else                { we store the first two sources of the other comps }
                                { we need this for the cycle determination later on }
              if (Components [Candidate]. Comp_Label = Other)
                    and (Other_Count < 2)
                then begin
                  Other_Count := Other_Count + 1;
                  Other_Sources [Other_Count] := Candidate
                end
        end;
      if Temp_List^. Next = Pair_List
        then Pair_List := Temp_List2;
      Temp_List^. Next := Temp_List2;
    end
    else with Temp_List^. Next^ do        { entry for this pair already exists }
      begin
        Counts [Components [Candidate]. Comp_Label]         { update stats }
          := Counts [Components [Candidate]. Comp_Label] + 1;
        if (Components [Candidate]. Comp_Label = Triangle)
          then Ring_Source := Candidate
          else
            if (Components [Candidate]. Comp_Label = Other) { as above ! }
                    and (Other_Count < 2)
              then begin
                Other_Count := Other_Count + 1;
```

```
                Other_Sources [Other_Count] := Candidate
              end
      end
end;

{-------------------------------------------------------------}
{- The separation pairs have been entered into the list.     -}
{- We now determine if there is a forbidden pair, or else    -}
{- the critical pairs. We discard all other separation pairs-}
{-------------------------------------------------------------}
procedure Analyse_Results;

var
  Deleted          : Boolean;    { if we have deleted the pair from our critical list }
  Ring_Bond_Count,               { count of all rings and bonds }
  Total_Count      : Byte;       { count of all triconnected components }
  Temp_List        : Separation_Pair_List_Ptr;

{-------------------------------------------------------------}
{- The pair is not critical or forbidden, so we discard it. -}
{-------------------------------------------------------------}
procedure Delete_Pair (Current_Posn : Separation_Pair_List_Ptr);

var
  Temp_List : Separation_Pair_List_Ptr;

begin
  Temp_List := Current_Posn^. Next;
  Current_Posn^. Next := Temp_List^. Next;    { remove from list }
  Deleted := true;
end;
```

```
{-------------------------------------------------------}
{- Begin of procedure Analyse_Results                  -}
{-------------------------------------------------------}
begin
  Finished := false;
  Base_Temp^. Next := Pair_List;   { dummy start of list to facilitate search }
  Temp_List := Base_Temp;
  Number_Critical := 0;
  while (not Finished) and (Temp_List^. Next <> nil) do
    begin
      Total_Count := 0;             { total number (x, y) - split components }
      Deleted := false;
      Ring_Bond_Count := 0;         { total number (x, y) rings and bonds }
      with Temp_List^. Next^ do     { determine the Ring Bond count }
        begin
          Ring_Bond_Count := Counts [Triangle] + Counts [Triple_Bond];
          Total_Count := Ring_Bond_Count + Counts [Other]
        end;
      case Total_Count of           { now check the results }
        3 : if Ring_Bond_Count = 0          { three others = forbidden }
              then begin
                Forbidden_Pair := true;
                Finished := true
              end
              else begin            { else placement is critical }
                Number_Critical := Number_Critical + 1;    { note that now critical }
                Graph^ [Temp_List^. Next^. Head]. Critical := true;
                Graph^ [Temp_List^. Next^. Tail]. Critical := true
              end;
        2 : if Ring_Bond_Count <> 0
              then Delete_Pair (Temp_List)  { two of any different kind is arbitrary }
              else begin            { two Others is critical }
                Number_Critical := Number_Critical + 1;
                Graph^ [Temp_List^. Next^. Head]. Critical := true;
                Graph^ [Temp_List^. Next^. Tail]. Critical := true
              end
        else if Total_Count > 3      { check for >= 4 separation pairs - forbidden }
               then begin
                 Forbidden_Pair := true;
                 Finished := true
               end
               else Delete_Pair (Temp_List)   { else sep. pair is arbitrary }
      end;
      if not Deleted      { proceed to next element only if no deletions }
        then Temp_List := Temp_List^. Next
        else Deleted := false
    end;
  Pair_List := Base_Temp^. Next    { note new head of critical list }
end;
```

```
{-------------------------------------------------------}
{- Main procedure to determine the critical pairs      -}
{-------------------------------------------------------}
begin
  Pair_List := nil;
  new (Base_Temp);      { to be used as dummy in above routines }
  Finished := false;
  Candidate := 1;       { start at first triconnected component }
  Check_Triangles;      { convert triangles to others if necessary }
  repeat                { for every component there is an (x, y) - split component }
    if (Components [Candidate]. Edges <> nil)   { if it was not merged with another to get triconnected }
      then begin
        Temp_Edge := Components [Candidate]. Edges;  { search thru for virtual edges }
        if Components [Candidate]. Comp_Label = Triple_Bond   { simple case! }
          then begin
            Head := Temp_Edge^. Head;
            Tail := Temp_Edge^. Tail;
            if Head > Tail
              then Swop (Head, Tail);
            Adjust_Pair_List (Head, Tail)           { and add to or adjust pair list }
          end
          else begin
            repeat                                  { for every edge }
              if Temp_Edge^. Origin <> Ordinary     { if virtual add (head, tail) split component }
                then begin
                  Head := Temp_Edge^. Head;
                  Tail := Temp_Edge^. Tail;
                  if Head > Tail
                    then Swop (Head, Tail);
                  Adjust_Pair_List (Head, Tail)     { and add to the list }
                end;
              Temp_Edge := Temp_Edge^. Next
            until Temp_Edge = nil
          end
      end;
    if not Finished
      then begin
        Candidate := Candidate + 1;
        Finished := (Candidate > Max_Candidates)
      end
  until Finished;
  Analyse_Results;          { now check for forbidden pairs,
                              search for critical pairs,
                              and discard the rest          }
end;
```

```
{-------------------------------------------------------------}
{- First cycle finding procedure.                            -}
{- Any cycle is extendible, so we merely trace out a face    -}
{-------------------------------------------------------------}
procedure Select_0_Cycle;

var
  Travel_Edge    : Edge_Ptr;
  Start_Vertex,
  Current_Vertex : Vertex_Ptr;

begin
  Init_Cycle (Initial_Cycle);
  Current_Vertex := 1;                              { select any undeleted vertex }
  While Graph^ [Current_Vertex]. Deleted do
    Current_Vertex := Current_Vertex + 1;

  Start_Vertex := Current_Vertex;                   { start vertex of the face }
  Add_to_Cycle (Initial_Cycle, Current_Vertex);     { build cycle }
  Graph^ [Current_Vertex]. Cyc_Clockwise := Graph^ [Current_Vertex]. Edges;  { note clockwise direction }
  Travel_Edge := Graph^ [Current_Vertex]. Edges^. Other_Edge;
  Current_Vertex := Graph^ [Current_Vertex]. Edges^. Vertex;                 { and proceed to next edge }
  repeat
    Graph^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge;                   { note anticlockwise direction }
    Add_to_Cycle (Initial_Cycle, Current_Vertex);
    Graph^ [Current_Vertex]. Cyc_Clockwise := Travel_Edge^. Previous;        { note clockwise direction }
    Current_Vertex := Travel_Edge^. Previous^. Vertex;
    Travel_Edge := Travel_Edge^. Previous^. Other_Edge
  until Current_Vertex = Start_Vertex;                                        { until face traced out }
  Graph^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge;                      { set last edge }
end;
```

```
{-----------------------------------------------------}
{- Select the single extendible cycle that has the required -}
{- properties - see Section 3.4 for more detail.           -}
{-----------------------------------------------------}
procedure Select_1_Cycle;

var
  Found_Component1,
  Found_Component2 : Boolean;      { we have to find a cycle through 2 components }
  Travel_Edge,
  Temp_Edge        : Edge_Ptr;
  Current_Vertex   : Vertex_Ptr;

{-----------------------------------------------------}
{- Note which vertices adjacent to separation pair vertex   -}
{- Head belongs to which component.                         -}
{-----------------------------------------------------}
procedure Setup_Component_Markers;

var
  Temp      : 1..2;
  Temp_Edge : Component_Edge_Ptr;

begin
  For Temp := 1 to Pair_List^. Other_Count do    { for every 3-connected component }
    begin
      Temp_Edge := Components [Pair_List^. Other_Sources [Temp]]. Edges;
      repeat
        if (Temp_Edge^. Head = Pair_List^. Head) or     { check for edge incident with Head }
           (Temp_Edge^. Tail = Pair_List^. Head)
          then
            if (Temp_Edge^. Tail = Pair_List^. Head)
              then Graph^ [Temp_Edge^. Head]. Component_ID    { note the component vertex belongs to }
                     := Pair_List^. Other_Sources [Temp]
              else Graph^ [Temp_Edge^. Tail]. Component_ID
                     := Pair_List^. Other_Sources [Temp];
        Temp_Edge := Temp_Edge^. Next
      until Temp_Edge = nil                              { until all edges done }
    end;
  Graph^ [Pair_List^. Head]. Component_ID := 0;         { Head belongs to no one }
  Graph^ [Pair_List^. Tail]. Component_ID := 0;
end;
```

```
{-------------------------------------------------------------}
{- This unit represents the Hopcroft & Tarjan algorithm.     -}
{- For a complete discussion of the method and Data          -}
{- structures used, please see Appendix A.                   -}
{-------------------------------------------------------------}
unit Hop_Algorithm;

interface

uses
  Planar_Defs,
  Planar_Miscellaneous,
  CRT;


procedure HopCroft_Tarjan;


implementation


{-------------------------------------------------------------}
{- The Depth First Search (DFS) is as per Even [4]. We need -}
{- perform a DFS in order to generate the lowpoints L1 and  -}
{- L2 which are essential for the algorithm to work.        -}
{-------------------------------------------------------------}
procedure DFS;

{-------------------------------------------------------------}
{- Returns the minimum of two numbers                        -}
{-------------------------------------------------------------}
function Min (x, y : Integer) : Integer;

begin
  if x < y
    then Min := x
    else Min := y;
end;
```

```
var
  Current_Label : 0..MaxInt;  { Current DFS number assigned }
  Temp_Edge   : Edge_Ptr;     { temporary variable      }
  Finished,                   { with the algorithm      }
  New_Vertex  : Boolean;      { indicates if a new      }
                              { label must be assigned }
  Temp_Vertex : Vertex_Ptr;   { temporary variable      }

begin
  For Temp_Vertex := 1 to Last_Vertex do     { Initialise the vertex }
    with Graph^ [Temp_Vertex] do
      begin
        Used := false;                       { have not reached the vertex yet }
        Number := 0;                         { no DFS number                   }
        Father := 0;                         { no father                       }
        L1 := 0;                             { no weightings                   }
        L2 := 0;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do            { for every incident edge         }
          begin
            Temp_Edge^. Used := False;       { we have not traversed the edge  }
            Temp_Edge^. Deleted := False;    { and it has not been directed    }
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;                         { Initial settings }
  New_Vertex := true;
  Temp_Vertex := 1;
  Current_Label := 0;
  repeat                  { until all vertices have been exhausted }
    repeat                  { until all edges of current vertex explored }
      {2}
      if New_Vertex   { if we need to assign initial labellings }
        then begin
          Current_Label := Current_Label + 1;
          Graph^ [Temp_Vertex]. Number := Current_Label;  { next label }
          Graph^ [Temp_Vertex]. L1 := Current_Label;      { default weightings }
          Graph^ [Temp_Vertex]. L2 := Current_Label;
        end;
      {3}
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      While (Temp_Edge <> Nil) and ((Temp_Edge^. Used) or (Temp_Edge^. Deleted)) do
        Temp_Edge := Temp_Edge^. Next;
      if (Temp_Edge <> Nil)              { we have an unexplored edge }
        then begin
          {4}
            Temp_Edge^. Used := True;                { note it is explored }
            Temp_Edge^. Other_Edge^. Deleted := true; { and direct the edge }
            with Graph^ [Temp_Edge^. Vertex] do
            if Number <> 0                           { The vertex has been visited before }
              then begin                             { so it is a back edge-adjust L1, L2 }
                if Number < Graph^ [Temp_Vertex]. L1
                  then begin
                    Graph^ [Temp_Vertex]. L2 := Graph^ [Temp_Vertex]. L1;
                    Graph^ [Temp_Vertex]. L1 := Number;
                  end
                  else
                    if Number > Graph^ [Temp_Vertex]. L1
```

```
                    then Graph^ [Temp_Vertex]. L2 :=
                           Min (Number, Graph^ [Temp_Vertex]. L2)
                    else;
                New_Vertex := false;
              end
              else begin                              { the edge is a tree edge }
                Father := Temp_Vertex;                { add to the tree         }
                Temp_Vertex := Temp_Edge^. Vertex;    { and move to the vertex  }
                New_Vertex := true;
              end;
        end
    until (Temp_Edge = Nil);      { if Temp_Edge = Nil then all edges explored }
                                  { and we need to backtrack to the father     }
    if Graph^ [Temp_Vertex]. Number = 1   { we are at the root - backtrack not possible }
      then
        {5} Finished := true
      else begin
        {6}
        with Graph^ [Graph^ [Temp_Vertex]. Father] do    { adjust father's weightings              }
          if Graph^ [Temp_Vertex]. L1 < L1               { based on kid's (more recent) weightings }
            then begin
              L2 := Min (Graph^ [Temp_Vertex]. L2, L1);
              L1 := Graph^ [Temp_Vertex]. L1
            end
          else
            if Graph^ [Temp_Vertex]. L1 = L1
              then L2 := Min (Graph^ [Temp_Vertex]. L2, L2)
              else L2 := Min (Graph^ [Temp_Vertex]. L1, L2);
        Temp_Vertex := Graph^ [Temp_Vertex]. Father;     { and backtrack }
        New_Vertex := false;
      end
  until Finished;
end;
```

```
{-------------------------------------------------------}
{- We reorder the edge lists of each vertex so that edges   -}
{- with a lower L1 weighting come first                     -}
{-                                                          -}
{- To do the actual sort, we place each edge in the graph   -}
{- into a bucket that represents its weighting. Then, we    -}
{- reconstruct the edge lists of each vertex by adding the  -}
{- edges from the highest bucket to the edges lists first.  -}
{- We add the edges to the front of the edge lists. Then we -}
{- proceed in order, considering lower buckets.             -}
{-------------------------------------------------------}
procedure ReOrder_Lists;

{-------------------------------------------------------}
{- Returns a weighting for the edge. Please see chapter 3   -}
{- for a justification of the weighting.                    -}
{-------------------------------------------------------}
function Phi (u, v : Vertex_Ptr) : Integer;

begin
  if Graph^ [v]. Number < Graph^ [u]. Number
    then Phi := 2 * Graph^ [v]. Number
    else if Graph^ [v]. L2 >= Graph^ [u]. Number
           then Phi := 2 * Graph^ [v]. L1
           else Phi := 2 * Graph^ [v]. L1 + 1
end;

var
  Bucket_Array : array [1..2 * Max_Vertices + 1] of Bucket_Ptr;
  Temp_Bucket,
  Temp_Bucket2 : Bucket_Ptr;
  Temp_Vertex  : Integer;
  Temp_Edge    : Edge_Ptr;

begin
  For Temp_Vertex := 1 to 2 * Last_Vertex + 1 do  { no edges in any bucket }
    Bucket_Array [Temp_Vertex] := Nil;
  For Temp_Vertex := 1 to Last_Vertex do          { for every vertex }
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while (Temp_Edge <> Nil) do                 { for each edge of that vertex }
        begin
          Temp_Edge^. Weight := Phi (Temp_Vertex, Temp_Edge^. Vertex); { get the weigth         }
          new (Temp_Bucket);                                           { get new bucket element }
          Temp_Bucket^. Data := Temp_Edge;
          Temp_Bucket^. Next := Bucket_Array [Temp_Edge^. Weight];     { and add element to the }
          Temp_Bucket^. Vertex := Temp_Vertex;                         { correct bucket         }
          Bucket_Array [Temp_Edge^. Weight] := Temp_Bucket;
          Temp_Edge := Temp_Edge^. Next                                { go to next edge        }
        end;
      Graph^ [Temp_Vertex]. Edges := Nil;                              { reset edge list        }
    end;
  For Temp_Vertex := 2 * Last_Vertex + 1 downto 1 do      { having sorted the edges   }
    begin                                                 { we add them back in order }
      Temp_Bucket := Bucket_Array [Temp_Vertex];
      while Temp_Bucket <> Nil do                 { while edges are in the bucket }
        begin
          Temp_Bucket^. Data^. Next := Graph^ [Temp_Bucket^. Vertex]. Edges; { add the edge }
```

```
          Graph^ [Temp_Bucket^. Vertex]. Edges := Temp_Bucket^. Data;
          Temp_Bucket2 := Temp_Bucket;                                  { remove the element }
          Temp_Bucket := Temp_Bucket^. Next;                            { from the bucket    }
          Dispose (Temp_Bucket2);
        end;
    end;
end;

{-------------------------------------------------------}
{- The main algorithm                                  -}
{-------------------------------------------------------}
procedure Hopcroft_Tarjan;

const
  Max_Edges = Max_Vertices * 3;

type                          { the following types are necessary to place the     }
                              { data structures onto the heap instead of the stack }
  Path_Array_Ptr = ^Path_Array;
  Path_Array     = Array [1..Max_Vertices] of Integer;   { stores the path emanating }
                                                         { from that vertex          }
  Stack_Array_Ptr = ^Stack_Array;
  Stack_Array     = Array [0..Max_Edges] of Integer;     { stack of fragments inside }
                                                         { and outside the cycle     }
                                                         { i.e. Inner and Outer      }
  Next_Array_Ptr  = ^Next_Array;
  Next_Array      = Array [-1..Max_Edges] of Integer;    { stores the next elements }
                                                         { of Inner and Outer       }
  f_Array_Ptr     = ^f_Array;
  f_Array         = Array [1..Max_Edges] of Integer;     { f(i) stores the last }
                                                         { vertex on the path i }
  B_Array_Ptr     = ^B_Array;
  B_Array         = Array [1..Max_Edges] of record       { if (x,y) is on B, then :           }
                      x, y : Integer                     { x - last entry of a block on Inner }
                    end;                                 { y - last entry of a block on Outer }

var
  Planar  : Boolean;          { if the graph is or is not }
  Current : Vertex_Ptr;      { Current vertex tested     }
  Free    : 1..Max_Edges;    { Next free space pointer onto stacks STACK and NEXT }

                             { The following variables are explained in the typedefs above }
  Path  : Path_Array_Ptr;
  Stack : Stack_Array_Ptr;
  Next  : Next_Array_Ptr;
  f     : f_Array_Ptr;
  B     : B_Array_Ptr;

  Path_to_Use,               { The current unused path                          }
  Start_Position,            { The start position of the current path           }
  x, y,                      { Temporary variables to address Inner and Outer   }
                             { stacks - values are from the Block stack         }
  B_Ptr : 0..Max_Edges;      { Temporary pointer to Block stack                 }


{-------------------------------------------------------}
{- Initialisation                                      -}
{-------------------------------------------------------}
```

```
procedure Initialise_Hopcroft_Tarjan;

var
  Temp : Vertex_Ptr;

begin
  Clrscr;
  DFS;                                  { Depth First Search }
  if Check_2_Connected
    then begin
      ReOrder_Lists;                        { and Reorder lists - very important }

                                        { create the data arrays on the heap }
      new (Next);
      new (f);
      new (Path);
      new (B);
      new (Stack);
      Next^ [-1] := 0;      { Outer stack empty - no fragments embedded }
      Next^ [0] := 0;       { Inner stack empty - no fragments embedded }
      Free := 1;            { first position available                  }
      Stack^ [0] := 0;      { end of stack marker                       }
      B_Ptr := 0;           { no blocks                                 }
      Path_to_Use := 0;     { no paths used                             }
      Start_Position := 0; { and start vertex not valid - no path       }
      Path^ [1] := 1;       { default first path                        }
      for Temp := 1 to Last_Vertex do
        Graph^ [Temp]. Used := false;  { note all vertices unvisited }
      Current := 1;                    { start at first vertex       }
      Planar := true
    end
    else Planar := false
end;

{-------------------------------------------------------------}
{- The recursive procedure that explores the fragments and  -}
{- embeds them.                                             -}
{-------------------------------------------------------------}
procedure PathFinder (Start_Vertex : Vertex_Ptr);

var
  Temp_Edge : Edge_Ptr;     { temporaray variable           }
  Outer : -1..Max_Edges;    { current pointer to Outer Stack }
  Inner,                    { current pointer to Inner Stack }
  Save  : 0.. Max_Edges;    { temporary variable             }

{-------------------------------------------------------------}
{- Returns the DFS number associated with a vertex. Note    -}
{- that every reference to a vertex must be done via the    -}
{- DFS number.                                              -}
{-------------------------------------------------------------}
function Num (x : Integer) : Integer;

begin
  Num := Graph^ [x]. Number
end;
```

```
{-------------------------------------------------------}
{- The current block on the top of the Block stack is read  -}
{- and returned.                                            -}
{-------------------------------------------------------}
procedure Read_TOS_B;

begin
  if B_Ptr <> 0              { if there is an element }
    then begin
      x := B^ [B_Ptr]. x;    { set the last attachment on the Inner stack }
      y := B^ [B_Ptr]. y;    { and the Outer stack                        }
    end
    else begin
      x := 0;                { no blocks on the stack, so return null }
      y := 0;
    end;
end;

{-------------------------------------------------------}
{- The Current path is not complete yet. Recursively call   -}
{- the algorithm with each adjacent unexplored vertex       -}
{- Then delete redundant attachments and blocks as we       -}
{- backtrack.                                               -}
{- For each new block, merge them onto the inside stack.    -}
{-------------------------------------------------------}
procedure Explore_Current_Path;

begin
  if Start_Position = 0               { if path is empty }
    then begin
      Start_Position := Start_Vertex;  { start a new path }
      Path_to_Use := Path_to_Use + 1   { and a new path number }
    end;
  Path^ [Temp_Edge^. Vertex] := Path_to_Use;  { note the path associated }
                                              { with this vertex         }
  PathFinder (Temp_Edge^. Vertex);     { recursively extend the path from new vertex }
  writeln ('Back at vertex ', Num(start_vertex));
  if not Planar
    then Exit;
  Read_TOS_B;          { update the Top of Stack values }

  while (B_Ptr <> 0)   { delete redundant blocks off the block stack }
      and ((Stack^ [x] >= Num (Start_Vertex)) or (x = 0))
        and ((Stack^ [y] >= Num (Start_Vertex)) or (y = 0)) do
    begin
      B_Ptr := B_Ptr - 1;
      Read_TOS_B;
    end;
  if Stack^ [x] >= Num (Start_Vertex)   { delete redundant attachments }
    then B^ [B_Ptr]. X := 0;            { off the Block Inner stack    }
  if Stack^ [y] >= Num (Start_Vertex)
    then B^ [B_Ptr]. Y := 0;            {.and off the Block Outer stack}

  Read_TOS_B;                           { update new block TOS }
  while (Next^ [-1] <> 0)                                  { delete redundant Outer and }
        and (Stack^ [Next^ [-1]] >= Num (Start_Vertex)) do  { redundant Inner elements   }
    Next^ [-1] := Next^ [Next^ [-1]];
  while (Next^ [0] <> 0)
```

```
        and (Stack^ [Next^ [0]] >= Num (Start_Vertex)) do
    Next^ [0] := Next^ [Next^ [0]];
  if Path^ [Temp_Edge^. Vertex] <> Path^ [Start_Vertex]     { now check if we gone back to }
    then begin                                              { the end of the current path. }
                                                            { if we have, then we need to  }
                                                            { embed the path - a new block }
      Inner := 0;             { lowest value on Inner for the new block }

      while (B_Ptr <> 0)      { while EOS marker not reached and an attachment }
                              { of the new paths fragments exists.             }
          and ((Stack^ [x] > Num (f^ [Path^ [Temp_Edge^. Vertex]])) or
               (Stack^ [y] > Num (f^ [Path^ [Temp_Edge^. Vertex]])))
          and (Stack^ [Next^ [-1]] <> 0) do
        begin
          if Stack^ [x] > Num (f^ [Path^ [Temp_Edge^. Vertex]])       { if it lies on the }
            then if Stack^ [y] > Num (f^ [Path^ [Temp_Edge^. Vertex]]) { inside and outside.    }
                   then begin
                     Planar := false;          { then non-planar - we cannot }
                     exit                      { embed the new path          }
                   end
                   else Inner := x             { else its on the inside - note  }
                                               { the lowest entry on the inside }
            else begin
              Save := Next^ [Inner];           { the block is on the outside }
              Next^ [Inner] := Next^ [-1];     { move it to the inside.      }
              Next^ [-1] := Next^ [y];
              Next^ [y] := Save;
              Inner := y                       { and note new lowest entry }
            end;
          B_Ptr := B_Ptr - 1;                  { delete this block }
          Read_TOS_B;                          { and get new block }
        end;
      if B_Ptr <> 0                    { now we have a single block for this }
                                       { fragment - we need to merge blocks  }
                                       { that interfere with this block.     }

        then while Stack^ [x] > Num (f^ [Path^ [Temp_Edge^. Vertex]]) do
               begin                   { while block lies on the inside in our way }
                 Inner := x;
                 B_Ptr := B_Ptr - 1;
                 Read_TOS_B;
               end;
      B_Ptr := B_Ptr - 1;              { wipe the current block }
      if x <> 0                        { if there is an element on the }
        then B_Ptr := B_Ptr + 1        { inside then restore the block }
        else if (Inner <> 0) or (y <> 0)  { else form new block }
               then begin
                 B_Ptr := B_Ptr + 1;
                 B^ [B_Ptr]. X := Inner;  { with correct lowest values }
                 B^ [B_Ptr]. Y := y;
                 Read_TOS_B;
               end;
      Next^ [-1] := Next^ [Next^ [-1]];   { and delete end_of_stack marker }
    end;
end;
```

```
{-------------------------------------------------------}
{- The first path in a new fragment is reached. We embed the-}
{- fragment on the inside of the cycle, merging blocks that -}
{- are placed by the embedding of the new path.            -}
{- Then we prepare to backtrack by adding an EOS marker so  -}
{- that we may distinguish between new and old blocks when  -}
{- we finish with this path.                               -}
{-------------------------------------------------------}
procedure Reached_End_of_Path;

begin  { the Current Edge is a back edge ---> the initial path is complete }
  if Start_Position = 0                  { could be a single back edge }
    then begin
      Path_to_Use := Path_to_Use + 1;
      Start_Position := Start_Vertex;
    end;
  f^ [Path_to_Use] := Temp_Edge^. Vertex; { note the finishing vertex }
  Inner := 0;                            { now embed the path on the inside }
  Outer := -1;
  Read_TOS_B;
  while (B_Ptr <> 0) and         { there is interlacing blocks }
        (
        ((Next^ [Inner] <> 0) and
         (Stack^ [Next^ [Inner]] > Num (Temp_Edge^. Vertex)))
     or ((Next^ [Outer] <> 0) and
         (Stack^ [Next^ [Outer]] > Num (Temp_Edge^. Vertex)))
        ) do
    begin
      if (B_Ptr <> 0) and (x <> 0) and (y <> 0)  { check if the block has }
        then begin                               { entries on both side   }
          if Stack^ [Next^ [Inner]] > Num (Temp_edge^. Vertex)
            then begin
              if Stack^ [Next^ [Outer]] > Num (Temp_edge^. Vertex)
                then begin                  { if the block interlaces on }
                  Planar := false;          { the inside and outside     }
                  exit;                     { then non-planar            }
                end;
              Save := Next^ [Outer];          { otherwise move the fragment   }
              Next^ [Outer] := Next^ [Inner]; { out of the way to the Outside }
              Next^ [Inner] := Save;          { of the cycle.                 }
              Save := Next^ [x];
              Next^ [x] := Next^ [y];
              Next^ [y] := Save;
              Inner := y;
              Outer := x;
            end
            else begin
              Inner := x;                   { the fragment is on the outside }
              Outer := y;                   { but now becomes part of the block }
            end
          end
        else if x <> 0                          { if there is only fragments on the inside }
               then begin
                 Save := Next^ [x];             { move them to the outside }
                 Next^ [x] := Next^ [Outer];
                 Next^ [Outer] := Next^ [Inner];
                 Next^ [Inner] := Save;
                 Outer := x;                    { and note new lowest outer attachment }
```

```
              end
              else if (y <> 0)                  { no moving necessary - becomes part of the  }
                    then Outer := y;            { new block - note new lowest out attachment }
      if B_Ptr <> 0
        then B_Ptr := B_Ptr - 1;                { and move to next block }
      Read_TOS_B;
    end;                { until no new blocks interlace }

  if (Num (f^ [Path^ [Start_Position]]) < Num (Temp_Edge^. Vertex))
        or (Start_Position = Temp_Edge^. Vertex)
    then begin          { avoid repeating attachments }
      if Inner = 0
        then Inner := Free;
      Stack^ [Free] := Num (Temp_Edge^. Vertex);   { add the new attachment }
      Next^ [Free] := Next^ [0];
      Next^ [0] := Free;
      Free := Free + 1;
    end;
  if Outer = -1
    then Outer := 0;
  if (Inner <> 0) or (Outer <> 0) or (Start_Vertex <> Start_Position)
    then begin
      B_Ptr := B_Ptr + 1;       { add the new block }
      B^ [B_Ptr]. x := Inner;
      B^ [B_Ptr]. y := Outer;
      Read_TOS_B;
    end;
  if Start_Vertex <> Start_Position  { if the edge is not a single back edge }
    then begin                       { then we need to explore the fragment  }
      Stack^ [Free] := 0;
      Next^ [Free] := Next^ [-1];    { add an end of stack marker }
      Next^ [-1] := Free;
      Free := Free + 1;
    end;
  Start_Position := 0;          { note the path is complete }
end;

{-------------------------------------------------------}
{- Start of main algorithm. We explore each edge.      -}
{-------------------------------------------------------}
begin
  writeln ('Recursing at vertex ', Num (start_vertex));
  Graph^ [Start_Vertex]. Used := true;          { we have visited this vertex }
  Temp_Edge := Graph^ [Start_Vertex]. Edges;
  while (Temp_Edge <> Nil) and (Temp_Edge^. Deleted) do  { get first valid edge }
    Temp_Edge := Temp_Edge^. Next;
  While Temp_Edge <> Nil do                             { for every edge from that vertex }
    begin
      if Num (Temp_Edge^. Vertex) > Num (Start_Vertex)   { if it is a tree edge }
        then Explore_Current_Path
        else Reached_End_of_Path;
      repeat
        Temp_Edge := Temp_Edge^. Next
      until (Temp_Edge = Nil) or (not Temp_Edge^. Deleted);   { and find a new edge to explore }
    end
end;
```

```
{-------------------------------------------------------}
{- Main procedure Select_1_Cycle.                      -}
{-------------------------------------------------------}
begin
  Setup_Component_Markers;    { mark all vertices adjacent to the vertex Head }
  case Pair_List^. Counts [Triangle] + Pair_List^. Counts [Triple_Bond] of
    0,1 : begin               { if only 0 or 1 triple bond or ring }
            Temp_Edge := Graph^ [Pair_List^. Head]. Edges;
            repeat                          { until correct cycle found }
              Found_Component1 := false;
              Found_Component2 := false;
              Init_Cycle (Initial_Cycle);
              Current_Vertex := Pair_List^. Head;
              Add_to_Cycle (Initial_Cycle, Current_Vertex);         { start a cycle }
              Graph^ [Current_Vertex]. Cyc_ClockWise := Temp_Edge;
              Current_Vertex := Temp_Edge^. Vertex;
              Travel_Edge := Temp_Edge^. Other_Edge;
              repeat                                                { trace out a face }
                Graph^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge;
                if Graph^ [Current_Vertex]. Component_ID <> 0
                  then if Pair_List^. Other_Sources [1]
                         = Graph^ [Current_Vertex]. Component_ID
                         then Found_Component1 := true
                         else Found_Component2 := true;
                Add_to_Cycle (Initial_Cycle, Current_Vertex);       { building up the cycle }
                Graph^ [Current_Vertex]. Cyc_ClockWise := Travel_Edge^. Previous;
                Current_Vertex := Travel_Edge^. Previous^. Vertex;
                Travel_Edge := Travel_Edge^. Previous^. Other_Edge;
              until Current_Vertex = Pair_List^. Head;              { until face complete }
              Graph^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge;
              Temp_Edge := Temp_Edge^. Next
            until (Found_Component1 and Found_Component2) { until cycle proceeds through both components }
          end;
    2 : begin
            { need the same as for case 1 ---> clockwise and anticlockwise }
            Temp_Edge := Graph^ [Pair_List^. Head]. Edges;
            repeat
              Found_Component1 := false;     { cycle needs to proceed through the one other component }
              Init_Cycle (Initial_Cycle);
              Setup_Component_Markers;
              Current_Vertex := Pair_List^. Head;
              Add_to_Cycle (Initial_Cycle, Current_Vertex);  { start a cycle }
              Graph^ [Current_Vertex]. Cyc_AntiClock := Temp_Edge;
              Current_Vertex := Temp_Edge^. Vertex;
              Travel_Edge := Temp_Edge^. Other_Edge;
              repeat
                Graph^ [Current_Vertex]. Cyc_ClockWise := Travel_Edge;
                Add_to_Cycle (Initial_Cycle, Current_Vertex);
                if Graph^ [Current_Vertex]. Component_ID <> 0
                  then Found_Component1 := true;
                Graph^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge^. Previous;
                Current_Vertex := Travel_Edge^. Previous^. Vertex;
                Travel_Edge := Travel_Edge^. Previous^. Other_Edge;
              until Current_Vertex = Pair_List^. Head;        { add until cycle complete }
              Graph^ [Current_Vertex]. Cyc_ClockWise := Travel_Edge;
              Temp_Edge := Temp_Edge^. Next
            until (Found_Component1)          { try all cycles until correct one }
          end;
```

```
    3 : begin
          Select_O_Cycle      { any cycle will do since three rings or bonds }
        end
    else begin
      writeln ('Fatal error - incorrect # bonds and triangles');
      halt
    end
  end
end;

{-------------------------------------------------------------}
{- We save the graph since we are about to modify it        -}
{-------------------------------------------------------------}
procedure Save_Graph;

var
  Temp_Graph : Graph_Ptr;
  Loop       : Vertex_Ptr;
  Temp_Edge2,
  Temp_Edge3,
  Temp_Edge  : Edge_Ptr;

begin
  new (Graph_Bup);                  { the backup graph }
  Graph_Bup^ := Graph^;             { block move the graph to a backup }
  Last_Vertex_Bup := Last_Vertex;
  For Loop := 1 to Last_Vertex do
    Graph_Bup^ [Loop]. Edges := nil; { reset original values }
  For Loop := 1 to Last_Vertex do   { duplicate the graph a vertex at a time }
    begin
      Temp_Edge := Graph^ [Loop]. Edges;
      while Temp_Edge <> nil do     { and all the edges }
        begin
          if Temp_Edge^. Vertex > Loop
            then begin
              new (Temp_Edge2);
              Temp_Edge2^ := Temp_Edge^;
              Temp_Edge2^. Next := Graph_Bup^ [Loop]. Edges;
              Graph_Bup^ [Loop]. Edges := Temp_Edge2;
              new (Temp_Edge3);
              Temp_Edge3^ := Temp_Edge^. Other_Edge^;
              Temp_Edge3^. Next := Graph_Bup^ [Temp_Edge^. Vertex]. Edges;
              Graph_Bup^ [Temp_Edge^. Vertex]. Edges := Temp_Edge3;
              Temp_Edge3^. Other_Edge := Temp_Edge2;
              Temp_Edge2^. Other_Edge := Temp_Edge3;
              end;
          Temp_Edge := Temp_Edge^. Next
        end
    end;
  Temp_Graph := Graph;
  Graph := Graph_Bup;
  Graph_Bup := Temp_Graph;
end;
```

```
{-------------------------------------------------------------}
{- Delete critical ring or bond components from the graph   -}
{-------------------------------------------------------------}
procedure Construct_G1;

var
  Temp_Pair      : Separation_Pair_List_Ptr;
  Base_Edge      : Edge_Ptr;
  Temp_Comp_Edge : Component_Edge_Ptr;

{-------------------------------------------------------------}
{- Delete the edge between From and Ref                      -}
{-------------------------------------------------------------}
procedure Delete (From, Ref : Vertex_Ptr);

var
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;

begin
  Base_Edge^. Next := Graph^ [From]. Edges;
  Temp_Edge := Base_Edge;
  while Temp_Edge^. Next^. Vertex <> Ref do
    Temp_Edge := Temp_Edge^. Next;
  Temp_Edge2 := Temp_Edge^. Next;
  Temp_Edge^. Next := Temp_Edge2^. Next;
  if Temp_Edge = Base_Edge
    then Graph^ [From]. Edges := Temp_Edge^. Next;
end;


{-------------------------------------------------------------}
{- Main procedure of graph G1                                -}
{-------------------------------------------------------------}
begin
  new (Base_Edge);
  Temp_Pair := Pair_List;
  repeat
    if Temp_Pair^. Counts [Triple_Bond] = 1              { delete the bond }
      then begin
        Delete (Temp_Pair^. Head, Temp_Pair^. Tail);
        Delete (Temp_Pair^. Tail, Temp_Pair^. Head)
      end
    else if Temp_Pair^. Counts [Triangle] = 1            { if exactly one then delete it }
          then begin
            Temp_Comp_Edge := Components [Temp_Pair^. Ring_Source]. Edges;
            repeat
              if (Temp_Comp_Edge^. Origin = Ordinary)    { remove all edges of ring }
                then begin
                  Delete (Temp_Comp_Edge^. Head, Temp_Comp_Edge^. Tail);
                  Delete (Temp_Comp_Edge^. Tail, Temp_Comp_Edge^. Head);
                end;
              Temp_Comp_Edge := Temp_Comp_Edge^. Next
            until Temp_Comp_Edge = nil
          end;
    Temp_Pair := Temp_Pair^. Next
  until Temp_Pair = nil;    { until we have donw every critical pair }
end;
```

```
{-------------------------------------------------------------}
{- The graph G2 is constructed. We add an extra vertex to   -}
{- the graph, and join every vertex that is a vertex of a   -}
{- critical separation pair to that vertex.                 -}
{- For the purposes of this section we define a critical    -}
{- vertex to be one that is a vertex of a critical          -}
{- separation pair.                                         -}
{-------------------------------------------------------------}
procedure Construct_G2;

var
  Temp_Edge,
  Temp_Edge2  : Edge_Ptr;
  Temp_Vertex : Vertex_Ptr;

begin
  Last_Vertex := Last_Vertex + 1;     { add an extra vertex }
  Number_Critical := 0;
  Graph^ [Last_Vertex]. Edges := nil; { no edges yet }
  For Temp_Vertex := 1 to Last_Vertex-1 do    { check if a vertex is critical }
    if Graph^ [Temp_Vertex]. Critical
      then begin
        Number_Critical := Number_Critical + 1;    { add to count }
        new (Temp_Edge);                           { and add the edge }
        new (Temp_Edge2);
        Temp_Edge^. Used := false;
        Temp_Edge^. Deleted := false;
        Temp_Edge2^. Used := false;
        Temp_Edge2^. Deleted := false;
        Temp_Edge^. Other_Edge := Temp_Edge2;
        Temp_Edge2^. Other_Edge := Temp_Edge;
        Temp_Edge^. Vertex := Last_Vertex;
        Temp_Edge2^. Vertex := Temp_Vertex;
        Temp_Edge^. Next := Graph^ [Temp_Vertex]. Edges;
        Graph^ [Temp_Vertex]. Edges := Temp_Edge;
        Temp_Edge2^. Next := Graph^ [Last_Vertex]. Edges;
        Graph^ [Last_Vertex]. Edges := Temp_Edge2;
      end
end;

{-------------------------------------------------------------}
{- We test planarity through a easy call to LEC algorithm   -}
{-------------------------------------------------------------}
procedure Test_Planarity_G2;

begin
  Lempel_Even_Cederbaum_Embedding
end;
```

```
{-------------------------------------------------------------}
{- We find the original facial cycle in G1 that has v in the-}
{- region.                                                   -}
{-------------------------------------------------------------}
procedure Select_v_Cycle;

var
  Temp_Cycle     : Cycle;
  New_Current_Vertex,
  Current_Vertex : Vertex_Ptr;
  Travel_Edge,
  Temp_Edge      : Edge_Ptr;

begin
  Temp_Edge := Graph^ [Last_Vertex]. Edges;
  Init_Cycle (Initial_Cycle);                    { we build a large cycle }
  repeat
    Init_Cycle (Temp_Cycle);                     { from a series of small cycles }
    Current_Vertex := Temp_Edge^. Vertex;
    Add_to_Cycle (Temp_Cycle, Current_Vertex);
    Travel_Edge := Temp_Edge^. Other_Edge;
    repeat                                       { traverse a cycle from v }
      if not Graph^ [Current_Vertex]. Critical
        then Add_to_Cycle (Temp_Cycle, Current_Vertex);
      if Travel_Edge^. Next = nil
        then begin
          New_Current_Vertex := Graph^ [Current_Vertex]. Edges^. Vertex;
          Travel_Edge := Graph^ [Current_Vertex]. Edges^. Other_Edge;
          Current_Vertex := New_Current_Vertex;
        end
        else begin
          Current_Vertex := Travel_Edge^. Next^. Vertex;
          Travel_Edge := Travel_Edge^. Next^. Other_Edge;
        end
    until Current_Vertex = Last_Vertex;          { until we get back to v }
    if Temp_Cycle. Length > 0
          then begin
        Reverse_Cycle (Temp_Cycle);              { must reverse to get proper order }
        Add_Cycles (Initial_Cycle, Temp_Cycle)   { add cycle less first and last edges }
      end;
    Temp_Edge := Temp_Edge^. Next
  until (Temp_Edge = nil);                       { repeat until every such face done }
  if Debug
    then Dump_cycle (Initial_Cycle)
end;
```

```
{-------------------------------------------------------------}
{- Used to add v to G, to ensure we obtain an embedding of  -}
{- G with v in one face.                                    -}
{-------------------------------------------------------------}
procedure Add_Vertex_Nbour;

var
  Temp_Element : Vertex_Ptr_Range;
  Temp_Edge,
  Temp_Edge2   : Edge_Ptr;
  Temp_Vertex  : Cycle_Element_Ptr;

begin
  Last_Vertex := Last_Vertex + 1;                    { add vertex v }
  Graph^ [Last_Vertex]. Edges := nil;
  Graph^ [Last_Vertex]. Deleted := false;
  Get_First_Element (Initial_Cycle, Temp_Vertex);
  For Temp_Element := 1 to Initial_Cycle. Length do  { for each vertex in bounding face }
    begin
      new (Temp_Edge);                    { add edge between v and vertex }
      new (Temp_Edge2);
      Temp_Edge^. Used := false;
      Temp_Edge^. Deleted := false;
      Temp_Edge2^. Used := false;
      Temp_Edge2^. Deleted := false;
      Temp_Edge^. Other_Edge := Temp_Edge2;
      Temp_Edge2^. Other_Edge := Temp_Edge;
      Temp_Edge^. Vertex := Last_Vertex;
      Temp_Edge2^. Vertex := Temp_Vertex^. Vertex;
      Temp_Edge^. Next := Graph^ [Temp_Vertex^. Vertex]. Edges;
      Graph^ [Temp_Vertex^. Vertex]. Edges := Temp_Edge;
      Temp_Edge2^. Next := Graph^ [Last_Vertex]. Edges;
      Graph^ [Last_Vertex]. Edges := Temp_Edge2;
      Get_Next_Element (Temp_Vertex)
    end
end;
```

```
{-------------------------------------------------------}
{- We have found an embedding with v in one face. Now remove-}
{- v from the graph.                                    -}
{-------------------------------------------------------}
procedure Remove_Vertex_Nbour;

var
  Temp_Edge2,
  Temp_Edge  : Edge_Ptr;
  Temp_Vertex: Vertex_Ptr;

begin
  Temp_Edge := Graph_G^ [Last_Vertex]. Edges;   { for every vertex adjacent to v }
  repeat
    Temp_Edge2 := Temp_Edge^. Other_Edge;
    Temp_Edge2^. Next^. Previous := Temp_Edge2^. Previous;
    Temp_Edge2^. Previous^. Next := Temp_Edge2^. Next;
    Graph_G^ [Temp_Edge^. Vertex]. Cyc_AntiClock := Temp_Edge2^. Next;    { note directions for drawing }
    Graph_G^ [Temp_Edge^. Vertex]. Cyc_Clockwise := Temp_Edge2^. Previous;
    Temp_Edge := Temp_Edge^. Next
  until Temp_Edge = Graph_G^ [Last_Vertex]. Edges;   { until finished }
  Last_Vertex := Last_Vertex - 1                     { note that v is deleted }
end;

{-------------------------------------------------------}
{- Debugging purposes only - we dump the current graph with -}
{- doubly linked adjacency lists.                       -}
{-------------------------------------------------------}
procedure Dump_Graph_Double;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge   : Edge_Ptr;
  Scr         : integer;

begin
  clrscr;
  Scr := 0;
  writeln ('The Graph is ');
  writeln;
  writeln ('Physical  Clock  Counter');
  For Temp_Vertex := 1 to Last_Vertex do  { for every vertex display the edges }
    if not Graph^ [Temp_Vertex]. Deleted
      then begin
        write (Temp_Vertex:4);
        Temp_Edge := Graph_G^ [Temp_Vertex]. Edges;
        write (Graph_G^ [Temp_Vertex]. Cyc_Clockwise^. Vertex:8,
               Graph_G^ [Temp_Vertex]. Cyc_AntiClock^. Vertex:8);
        write (' ':4);
        repeat
          write ('(', Temp_Edge^. Vertex,',', Temp_Edge^. Other_Edge^. Vertex,')', '  ');
          Temp_Edge := Temp_Edge^. Next
        until Temp_Edge = Graph_G^ [Temp_Vertex]. Edges;
        writeln;
        Scr := Scr + 1;
        if Scr = 20     { check for full screen and wait if necessary }
          then begin
            Prompt;
```

```
            Scr := 0;
          end;
      end;
  writeln ('and now back to fun! ....');
  writeln;
  prompt;
end;

{-------------------------------------------------------------}
{- We convert  singly linked adjacency list to doubly linked-}
{- to facilitate easy removal and deletion                   -}
{-------------------------------------------------------------}
procedure Set_Up_Double_Lists;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge,
  Previous    : Edge_Ptr;

begin
  For Temp_Vertex := 1 to Last_Vertex do
    if not Graph_G^ [Temp_Vertex]. Deleted
      then begin                                        { ignore deleted vertices }
        Temp_Edge := Graph_G^ [Temp_Vertex]. Edges;
        Previous := nil;
        while Temp_Edge^. Next <> nil do                { now add extra references }
          begin
            Temp_Edge^. Previous := Previous;
            Previous := Temp_Edge;
            Temp_Edge := Temp_Edge^. Next
          end;
        Temp_Edge^. Previous := Previous;               { last case is special case }
        Temp_Edge^. Next := Graph_G^ [Temp_Vertex]. Edges;
        Graph_G^ [Temp_Vertex]. Edges^. Previous := Temp_Edge
      end
end;
```

```
{-------------------------------------------------------------}
{- Main procedure Convex Test and Draw the graph             -}
{-------------------------------------------------------------}
procedure Convex_Test_Graph;

begin
  Initialise;
  Lempel_Even_Cederbaum_Embedding;          { find an embedding }
  if (not Planar) or (not Is_2_Connected)
    then
      if not Planar
        then begin
          writeln ('The Graph is not planar');
          prompt
        end
      else prompt
    else begin
      Save_Graph;                       { preserve original graph }
      Graph_G := Graph;
      Remove_Degree_2_Vertices;
      Find_Separation_Pairs;            { find pairs and triconnected components }
      Determine_Critical_Pairs;         { determine which pairs are critical }
      if Not Forbidden_Pair
        then case Number_Critical of
               0 : begin                { we can choose any cycle }
                     TextColor (Red);
                     writeln ('All cycles are extendible');
                     writeln;
                     TextColor (Yellow);
                     Restore_Graph;              { restore old graph }
                     Graph_G := Graph;
                     Set_up_Double_Lists;        { doubly linked for convenience }
                     Select_0_Cycle;             { find any cycle }
                   end;
               1 : begin
                     TextColor (Red);
                     writeln ('Selecting the single extendible cycle');
                     writeln;
                     TextColor (Yellow);
                     Restore_Graph;              { as above }
                     Graph_G := Graph;
                     Set_up_Double_Lists;        {    "     }
                     Select_1_Cycle;             { this time find specific cycle }
                   end
             else begin        { the tricky case! }
               Construct_G1;                     { delete rings, bonds as necessary }
               Construct_G2;                     { add extra vertex }
               Test_Planarity_G2;                { find extendible cycle in G1 }
               if not Planar
                 then begin     { failure }
                   TextColor (Red);
                   writeln ('G has no convex drawing');
                   Extendible_Cycle := false
                 end
                 else begin
                   TextColor (Red);
                   writeln ('Selecting the v-cycle');
                   writeln;
```

```
                    TextColor (Yellow);
                    Select_v_Cycle;
                    Restore_Graph;              { as above }
                    Save_Graph;
                    Add_Vertex_Nbour;           { add v to G as well as G1! }
                    Lempel_Even_Cederbaum_Embedding; { find embedding }
                    if Planar
                      then begin
                        Graph_G := Graph;
                        Set_Up_Double_Lists;
                        Remove_Vertex_Nbour;       { now find embedding of G by removing v }
                      end
                      else Extendible_Cycle := false
                  end
              end
            end
        else Extendible_Cycle := false;
      if Extendible_Cycle
        then begin
          Writeln ('We will now proceed to draw the graph');
          prompt;
          Convex_Drawing     { at last ----> draw it! }
        end
        else begin
          writeln ('A forbidden pair has been detected - no convex drawing possible');
          prompt
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit will perform the actaul drawing algorithm.    -}
{-------------------------------------------------------}
unit Convex_Draw;

interface


procedure Convex_Drawing;     { single procedure to extend a convex cycle }


implementation

uses
  Planar_Defs,                { main type defs and global variables }
  Planar_Miscellaneous,       { service procedures }
  Convex_Globals,             { globals for convex testing and drawing routines }
  CRT,
  Graph;                      { TURBOPascal graphics unit }

Const
  Up            = True;       { positive approach }
  Down          = false;
  Border_Offset = 30;         { offset from screen edges }

var
  Global_Y        : real;     { used during multiple passes of the same drawing routine }
  Min_X, Min_Y,
  Max_X, Max_Y    : Coords;   { screen coords }


{-------------------------------------------------------}
{ Swops two Coords                                      -}
{-------------------------------------------------------}
procedure Swop (var a, b : Coords);

var
  Temp : Coords;

begin
  Temp := a;
  a := b;
  b := Temp
end;

{-------------------------------------------------------}
{ Finds floor of a scalar                               -}
{-------------------------------------------------------}
function Floor (Num : Coords) : Coords;

begin
  Floor := Num div 2
end;
```

```
{-------------------------------------------------------}
{ Finds ceiling of a scalar                             -}
{-------------------------------------------------------}
function Ceiling (Num : Coords) : Coords;

begin
  if Num mod 2 <> 0
    then Ceiling := (Num div 2) + 1
    else Ceiling := Num div 2
end;

{-------------------------------------------------------}
{ Translate a rotation of theta radians for X-coord.    -}
{-------------------------------------------------------}
function X_Trans (Theta : real; X_Prime, Y_Prime : Coords) : Coords;

var
  X_Transa : Coords;

begin
  X_Transa := Round (X_Prime * Cos (Theta)
                   + Y_Prime * Sin (Theta));
  X_Trans := X_Transa;
end;

{-------------------------------------------------------}
{ Translate a rotation of theta radians for Y-coord.    -}
{-------------------------------------------------------}
function Y_Trans (Theta : real; X_Prime, Y_Prime : Coords) : Coords;

var
  Y_Transa : Coords;

begin
  Y_Transa := Round (- X_Prime * Sin (Theta)
                    + Y_Prime * Cos (Theta));
  Y_Trans := Y_Transa
end;

{-------------------------------------------------------}
{ Translate back from rotation of theta radians for X-coord.-}
{-------------------------------------------------------}
function Reverse_X_Trans (Theta, X_Prime, Y_Prime : real) : Coords;

begin
  Reverse_X_Trans := Round (X_Prime * Cos (Theta)
                    - Y_Prime * Sin (Theta));
end;

{-------------------------------------------------------}
{ Translate back from rotation of theta radians for Y-coord.-}
{-------------------------------------------------------}
function Reverse_Y_Trans (Theta, X_Prime, Y_Prime : real) : Coords;

begin
  Reverse_Y_Trans := Round (X_Prime * Sin (Theta)
                           + Y_Prime * Cos (Theta));
end;
```

```
{-------------------------------------------------------}
{- Calculate the angle theta between two vertices      -}
{- We have already ensured the two x values are not zero -}
{-------------------------------------------------------}
Function Calculate_Theta (Vert1, Vert2 : Vertex) : real;

var
  Delta_X,
  Delta_Y  : Integer;

begin
  Delta_Y := Vert2. Y - Vert1. Y;
  Delta_X := Vert2. X - Vert1. X;
  Calculate_Theta := Arctan (Delta_Y / Delta_X)
end;

{-------------------------------------------------------}
{- Initialise graphics, print border and get screen size -}
{-------------------------------------------------------}
procedure Init_Graphics;

var
  Val_Str     : string;
  Temp        : Vertex_Ptr;
  GraphDriver,
  GraphMode   : Integer;

begin
  For Temp := 1 to Last_Vertex do
    Graph_G^ [Temp]. Number := Temp;
  GraphDriver := Detect;                    { work for any monitor }
  Initgraph  (GraphDriver, GraphMode, '');  { initialise for graphics }
  SetColor (GetMaxColor);
  Max_X := GetMaxX;                         { get screen coords }
  Max_Y := GetMaxY;
  Min_X := 0;
  Min_Y := 0;

  Line (Min_X, Min_Y, Min_X, Max_Y);        { draw border around screen }
  Line (Min_X, Min_Y, Max_X, Min_Y);
  Line (Max_X, Min_Y, Max_X, Max_Y);
  Line (Min_X, Max_Y, Max_X, Max_Y);

  SetTextStyle (DefaultFont, HorizDir, 1)
end;
```

```
{-------------------------------------------------------}
{- Draw the initial cycle. We could merely draw a circle and-}
{- place the vertices at regularly spaced intervals on the  -}
{- circle, but because of screen shape (for EGA 640x320),   -}
{- we prefer to use a diamond or slightly elliptical shape. -}
{-                                                          -}
{- This approach, although more tedious does allow a large  -}
{- degree of freedom over the choice of initial cycle layout-}
{-------------------------------------------------------}
procedure Draw_Initial_Cycle;

var
  First,
  Second,
  Start_Vertex   : Cycle_Element_Ptr;
  Offset,
  First_Y,
  Second_Y,
  End_X, Start_X : Coords;

{-------------------------------------------------------}
{- We place two vertices on the LHS of the screen          -}
{-------------------------------------------------------}
procedure Place_Two_Vertices;

var
  Current,
  Second : Cycle_Element_Ptr;

begin
  Get_First_Element (Initial_Cycle, Current);       { get first two elements }
  Second := Current;
  Get_Next_Element (Second);
  Graph_G^ [Second^. Vertex]. X := Min_X + Border_Offset;
  Graph_G^ [Current^. Vertex]. X := Min_X + Border_Offset;
  Graph_G^ [Second^. Vertex]. Y := ((Min_Y + Max_Y) div 2) + Offset;
  Graph_G^ [Current^. Vertex]. Y := ((Min_Y + Max_Y) div 2) - Offset;
  Draw_Vertex (Second^. Vertex, Graph_G^ [Second^. Vertex]);    { do the drawing }
  Draw_Vertex (Current^. Vertex, Graph_G^ [Current^. Vertex]);
  Start_X := Graph_G^ [Current^. Vertex]. X;
  First_Y := Graph_G^ [Second^. Vertex]. Y;
  Second_Y := Graph_G^ [Current^. Vertex]. Y;
  Graph_G^ [Current^. Vertex]. Queued := true;
  Graph_G^ [Second^. Vertex]. Queued := true;
  Start_Vertex := Second;
  Get_Next_Element (Start_Vertex)           { and move onto the third vertex }
end;
```

```
{-------------------------------------------------------}
{- Place two vertices on RHS of screen.                -}
{-------------------------------------------------------}
procedure Place_Another_Two_Vertices;

var
  Loop              : Vertex_Ptr;
  Saved_Start_Vertex,
  Half_Way2,
  Half_Way          : Cycle_Element_Ptr;

begin
  Saved_Start_Vertex := Start_Vertex;
  For Loop := 3 to (Initial_Cycle. Length div 2) do
    Get_Next_Element (Start_Vertex);          { move along cycle to get to middle elements }
  Half_Way := Start_Vertex;
  Half_Way2 := Start_Vertex;
  Get_Next_Element (Half_Way2);
  Graph_G^ [Half_Way^. Vertex]. X := Max_X - Border_Offset;
  Graph_G^ [Half_Way2^. Vertex]. X := Max_X - Border_Offset;
  Graph_G^ [Half_Way^. Vertex]. Y :=
                        ((Min_Y + Max_Y) div 2) + (Offset);
  Graph_G^ [Half_Way2^. Vertex]. Y :=
                        ((Min_Y + Max_Y) div 2) - (Offset);
  Draw_Vertex (Half_Way^. Vertex,                    { draw the two middle vertices }
                Graph_G^ [Half_Way^. Vertex]);
  Draw_Vertex (Half_Way2^. Vertex,
                Graph_G^ [Half_Way2^. Vertex]);
  End_X := Graph_G^ [Half_Way^. Vertex]. X;
  Graph_G^ [Half_Way^. Vertex]. Queued := true;
  Graph_G^ [Half_Way2^. Vertex]. Queued := true;
  Start_Vertex := Saved_Start_Vertex
end;
```

```
{-------------------------------------------------------}
{- Now we proceed to draw half of the polygon between       -}
{-     Start_X and End_X, Start_Y_Coord and Boundary_n embedding of  -}
{- G with v in one face.                                    -}
{-------------------------------------------------------}
procedure Add_Vertex_Nbour;

var
  Temp_Element : Vertex_Ptr_Range;
  Temp_Edge,
  Temp_Edge2   : Edge_Ptr;
  Temp_Vertex  : Cycle_Element_Ptr;

begin
  Last_Vertex := Last_Vertex + 1;                    { add vertex v }
  Graph^ [Last_Vertex]. Edges := nil;
  Graph^ [Last_Vertex]. Deleted := false;
  Get_First_Element (Initial_Cycle, Temp_Vertex);
  For Temp_Element := 1 to Initial_Cycle. Length do  { for each vertex in bounding face }
    begin
      new (Temp_Edge);                   { add edge between v and vertex }
      new (Temp_Edge2);
      Temp_Edge^. Used := false;
      Temp_Edge^. Deleted := false;
      Temp_Edge2^. Used := false;
      Temp_Edge2^. Deleted := false;
      Temp_Edge^. Other_Edge := Temp_Edge2;
      Temp_Edge2^. Other_Edge := Temp_Edge;
      Temp_Edge^. Vertex := Last_Vertex;
      Temp_Edge2^. Vertex := Temp_Vertex^. Vertex;
      Temp_Edge^. Next := Graph^ [Temp_Vertex^. Vertex]. Edges;
      Graph^ [Temp_Vertex^. Vertex]. Edges := Temp_Edge;
      Temp_Edge2^. Next := Graph^ [Last_Vertex]. Edges;
      Graph^ [Last_Vertex]. Edges := Temp_Edge2;
      Get_Next_Element (Temp_Vertex)
    end
end;
```

```
{-------------------------------------------------------}
{- We have found an embedding with v in one face. Now remove-}
{- v from the graph.                                    -}
{-------------------------------------------------------}
procedure Remove_Vertex_Nbour;

var
  Temp_Edge2,
  Temp_Edge  : Edge_Ptr;
  Temp_Vertex: Vertex_Ptr;

begin
  Temp_Edge := Graph_G^ [Last_Vertex]. Edges;   { for every vertex adjacent to v }
  repeat
    Temp_Edge2 := Temp_Edge^. Other_Edge;
    Temp_Edge2^. Next^. Previous := Temp_Edge2^. Previous;
    Temp_Edge2^. Previous^. Next := Temp_Edge2^. Next;
    Graph_G^ [Temp_Edge^. Vertex]. Cyc_AntiClock := Temp_Edge2^. Next;    { note directions for drawing }
    Graph_G^ [Temp_Edge^. Vertex]. Cyc_Clockwise := Temp_Edge2^. Previous;
    Temp_Edge := Temp_Edge^. Next
  until Temp_Edge = Graph_G^ [Last_Vertex]. Edges;   { until finished }
  Last_Vertex := Last_Vertex - 1                     { note that v is deleted }
end;

{-------------------------------------------------------}
{- Debugging purposes only - we dump the current graph with -}
{- doubly linked adjacency lists.                       -}
{-------------------------------------------------------}
procedure Dump_Graph_Double;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge   : Edge_Ptr;
  Scr         : integer;

begin
  clrscr;
  Scr := 0;
  writeln ('The Graph is ');
  writeln;
  writeln ('Physical  Clock  Counter');
  For Temp_Vertex := 1 to Last_Vertex do  { for every vertex display the edges }
    if not Graph^ [Temp_Vertex]. Deleted
      then begin
        write (Temp_Vertex:4);
        Temp_Edge := Graph_G^ [Temp_Vertex]. Edges;
        write (Graph_G^ [Temp_Vertex]. Cyc_Clockwise^. Vertex:8,
               Graph_G^ [Temp_Vertex]. Cyc_AntiClock^. Vertex:8);
        write (' ':4);
        repeat
          write ('(', Temp_Edge^. Vertex,',', Temp_Edge^. Other_Edge^. Vertex,')', '  ');
          Temp_Edge := Temp_Edge^. Next
        until Temp_Edge = Graph_G^ [Temp_Vertex]. Edges;
        writeln;
        Scr := Scr + 1;
        if Scr = 20     { check for full screen and wait if necessary }
          then begin
            Prompt;
```

```
          Scr := 0;
        end;
    end;
  writeln ('and now back to fun! ....');
  writeln;
  prompt;
end;

{-------------------------------------------------------}
{- We convert  singly linked adjacency list to doubly linked-}
{- to facilitate easy removal and deletion                -}
{-------------------------------------------------------}
procedure Set_Up_Double_Lists;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge,
  Previous    : Edge_Ptr;

begin
  For Temp_Vertex := 1 to Last_Vertex do
    if not Graph_G^ [Temp_Vertex]. Deleted
      then begin                                    { ignore deleted vertices }
        Temp_Edge := Graph_G^ [Temp_Vertex]. Edges;
        Previous := nil;
        while Temp_Edge^. Next <> nil do            { now add extra references }
          begin
            Temp_Edge^. Previous := Previous;
            Previous := Temp_Edge;
            Temp_Edge := Temp_Edge^. Next
          end;
        Temp_Edge^. Previous := Previous;           { last case is special case }
        Temp_Edge^. Next := Graph_G^ [Temp_Vertex]. Edges;
        Graph_G^ [Temp_Vertex]. Edges^. Previous := Temp_Edge
      end
end;
```

```
{-------------------------------------------------------}
{- Main procedure Convex Test and Draw the graph       -}
{-------------------------------------------------------}
procedure Convex_Test_Graph;

begin
  Initialise;
  Lempel_Even_Cederbaum_Embedding;          { find an embedding }
  if (not Planar) or (not Is_2_Connected)
    then
      if not Planar
        then begin
          writeln ('The Graph is not planar');
          prompt
        end
      else prompt
    else begin
      Save_Graph;                    { preserve original graph }
      Graph_G := Graph;
      Remove_Degree_2_Vertices;
      Find_Separation_Pairs;         { find pairs and triconnected components }
      Determine_Critical_Pairs;      { determine which pairs are critical }
      if Not Forbidden_Pair
        then case Number_Critical of
               0 : begin             { we can choose any cycle }
                     TextColor (Red);
                     writeln ('All cycles are extendible');
                     writeln;
                     TextColor (Yellow);
                     Restore_Graph;              { restore old graph }
                     Graph_G := Graph;
                     Set_up_Double_Lists;        { doubly linked for convenience }
                     Select_0_Cycle;             { find any cycle }
                   end;
               1 : begin
                     TextColor (Red);
                     writeln ('Selecting the single extendible cycle');
                     writeln;
                     TextColor (Yellow);
                     Restore_Graph;              { as above }
                     Graph_G := Graph;
                     Set_up_Double_Lists;        {    "     }
                     Select_1_Cycle;             { this time find specific cycle }
                   end
             else begin        { the tricky case! }
               Construct_G1;                     { delete rings, bonds as necessary }
               Construct_G2;                     { add extra vertex }
               Test_Planarity_G2;                { find extendible cycle in G1 }
               if not Planar
                 then begin     { failure }
                   TextColor (Red);
                   writeln ('G has no convex drawing');
                   Extendible_Cycle := false
                 end
                 else begin
                   TextColor (Red);
                   writeln ('Selecting the v-cycle');
                   writeln;
```

```
                    TextColor (Yellow);
                    Select_v_Cycle;
                    Restore_Graph;              { as above }
                    Save_Graph;
                    Add_Vertex_Nbour;           { add v to G as well as G1! }
                    Lempel_Even_Cederbaum_Embedding; { find embedding }
                    if Planar
                      then begin
                        Graph_G := Graph;
                        Set_Up_Double_Lists;
                        Remove_Vertex_Nbour;       { now find embedding of G by removing v }
                      end
                      else Extendible_Cycle := false
                  end
              end
            end
        else Extendible_Cycle := false;
      if Extendible_Cycle
        then begin
          Writeln ('We will now proceed to draw the graph');
          prompt;
          Convex_Drawing     { at last ----> draw it! }
        end
        else begin
          writeln ('A forbidden pair has been detected - no convex drawing possible');
          prompt
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit will perform the actaul drawing algorithm.    -}
{-------------------------------------------------------}
unit Convex_Draw;

interface


procedure Convex_Drawing;     { single procedure to extend a convex cycle }


implementation

uses
  Planar_Defs,                { main type defs and global variables }
  Planar_Miscellaneous,       { service procedures }
  Convex_Globals,             { globals for convex testing and drawing routines }
  CRT,
  Graph;                      { TURBOPascal graphics unit }

Const
  Up            = True;       { positive approach }
  Down          = false;
  Border_Offset = 30;         { offset from screen edges }

var
  Global_Y        : real;     { used during multiple passes of the same drawing routine }
  Min_X, Min_Y,
  Max_X, Max_Y    : Coords;   { screen coords }


{-------------------------------------------------------}
{ Swops two Coords                                      -}
{-------------------------------------------------------}
procedure Swop (var a, b : Coords);

var
  Temp : Coords;

begin
  Temp := a;
  a := b;
  b := Temp
end;

{-------------------------------------------------------}
{ Finds floor of a scalar                               -}
{-------------------------------------------------------}
function Floor (Num : Coords) : Coords;

begin
  Floor := Num div 2
end;
```

```
{-------------------------------------------------------}
{ Finds ceiling of a scalar                             -}
{-------------------------------------------------------}
function Ceiling (Num : Coords) : Coords;

begin
  if Num mod 2 <> 0
    then Ceiling := (Num div 2) + 1
    else Ceiling := Num div 2
end;

{-------------------------------------------------------}
{ Translate a rotation of theta radians for X-coord.    -}
{-------------------------------------------------------}
function X_Trans (Theta : real; X_Prime, Y_Prime : Coords) : Coords;

var
  X_Transa : Coords;

begin
  X_Transa := Round (X_Prime * Cos (Theta)
                   + Y_Prime * Sin (Theta));
  X_Trans := X_Transa;
end;

{-------------------------------------------------------}
{ Translate a rotation of theta radians for Y-coord.    -}
{-------------------------------------------------------}
function Y_Trans (Theta : real; X_Prime, Y_Prime : Coords) : Coords;

var
  Y_Transa : Coords;

begin
  Y_Transa := Round (- X_Prime * Sin (Theta)
                    + Y_Prime * Cos (Theta));
  Y_Trans := Y_Transa
end;

{-------------------------------------------------------}
{ Translate back from rotation of theta radians for X-coord.-}
{-------------------------------------------------------}
function Reverse_X_Trans (Theta, X_Prime, Y_Prime : real) : Coords;

begin
  Reverse_X_Trans := Round (X_Prime * Cos (Theta)
                    - Y_Prime * Sin (Theta));
end;

{-------------------------------------------------------}
{ Translate back from rotation of theta radians for Y-coord.-}
{-------------------------------------------------------}
function Reverse_Y_Trans (Theta, X_Prime, Y_Prime : real) : Coords;

begin
  Reverse_Y_Trans := Round (X_Prime * Sin (Theta)
                           + Y_Prime * Cos (Theta));
end;
```

```
{-------------------------------------------------------}
{- Calculate the angle theta between two vertices      -}
{- We have already ensured the two x values are not zero -}
{-------------------------------------------------------}
Function Calculate_Theta (Vert1, Vert2 : Vertex) : real;

var
  Delta_X,
  Delta_Y  : Integer;

begin
  Delta_Y := Vert2. Y - Vert1. Y;
  Delta_X := Vert2. X - Vert1. X;
  Calculate_Theta := Arctan (Delta_Y / Delta_X)
end;

{-------------------------------------------------------}
{- Initialise graphics, print border and get screen size -}
{-------------------------------------------------------}
procedure Init_Graphics;

var
  Val_Str     : string;
  Temp        : Vertex_Ptr;
  GraphDriver,
  GraphMode   : Integer;

begin
  For Temp := 1 to Last_Vertex do
    Graph_G^ [Temp]. Number := Temp;
  GraphDriver := Detect;                    { work for any monitor }
  Initgraph  (GraphDriver, GraphMode, '');  { initialise for graphics }
  SetColor (GetMaxColor);
  Max_X := GetMaxX;                         { get screen coords }
  Max_Y := GetMaxY;
  Min_X := 0;
  Min_Y := 0;

  Line (Min_X, Min_Y, Min_X, Max_Y);        { draw border around screen }
  Line (Min_X, Min_Y, Max_X, Min_Y);
  Line (Max_X, Min_Y, Max_X, Max_Y);
  Line (Min_X, Max_Y, Max_X, Max_Y);

  SetTextStyle (DefaultFont, HorizDir, 1)
end;
```

```
{-------------------------------------------------------}
{- Draw the initial cycle. We could merely draw a circle and-}
{- place the vertices at regularly spaced intervals on the  -}
{- circle, but because of screen shape (for EGA 640x320),   -}
{- we prefer to use a diamond or slightly elliptical shape. -}
{-                                                          -}
{- This approach, although more tedious does allow a large  -}
{- degree of freedom over the choice of initial cycle layout-}
{-------------------------------------------------------}
procedure Draw_Initial_Cycle;

var
  First,
  Second,
  Start_Vertex   : Cycle_Element_Ptr;
  Offset,
  First_Y,
  Second_Y,
  End_X, Start_X : Coords;

{-------------------------------------------------------}
{- We place two vertices on the LHS of the screen          -}
{-------------------------------------------------------}
procedure Place_Two_Vertices;

var
  Current,
  Second : Cycle_Element_Ptr;

begin
  Get_First_Element (Initial_Cycle, Current);       { get first two elements }
  Second := Current;
  Get_Next_Element (Second);
  Graph_G^ [Second^. Vertex]. X := Min_X + Border_Offset;
  Graph_G^ [Current^. Vertex]. X := Min_X + Border_Offset;
  Graph_G^ [Second^. Vertex]. Y := ((Min_Y + Max_Y) div 2) + Offset;
  Graph_G^ [Current^. Vertex]. Y := ((Min_Y + Max_Y) div 2) - Offset;
  Draw_Vertex (Second^. Vertex, Graph_G^ [Second^. Vertex]);    { do the drawing }
  Draw_Vertex (Current^. Vertex, Graph_G^ [Current^. Vertex]);
  Start_X := Graph_G^ [Current^. Vertex]. X;
  First_Y := Graph_G^ [Second^. Vertex]. Y;
  Second_Y := Graph_G^ [Current^. Vertex]. Y;
  Graph_G^ [Current^. Vertex]. Queued := true;
  Graph_G^ [Second^. Vertex]. Queued := true;
  Start_Vertex := Second;
  Get_Next_Element (Start_Vertex)           { and move onto the third vertex }
end;
```

```
{-------------------------------------------------------}
{- Place two vertices on RHS of screen.                -}
{-------------------------------------------------------}
procedure Place_Another_Two_Vertices;

var
  Loop              : Vertex_Ptr;
  Saved_Start_Vertex,
  Half_Way2,
  Half_Way          : Cycle_Element_Ptr;

begin
  Saved_Start_Vertex := Start_Vertex;
  For Loop := 3 to (Initial_Cycle. Length div 2) do
    Get_Next_Element (Start_Vertex);          { move along cycle to get to middle elements }
  Half_Way := Start_Vertex;
  Half_Way2 := Start_Vertex;
  Get_Next_Element (Half_Way2);
  Graph_G^ [Half_Way^. Vertex]. X := Max_X - Border_Offset;
  Graph_G^ [Half_Way2^. Vertex]. X := Max_X - Border_Offset;
  Graph_G^ [Half_Way^. Vertex]. Y :=
                       ((Min_Y + Max_Y) div 2) + (Offset);
  Graph_G^ [Half_Way2^. Vertex]. Y :=
                       ((Min_Y + Max_Y) div 2) - (Offset);
  Draw_Vertex (Half_Way^. Vertex,                    { draw the two middle vertices }
                Graph_G^ [Half_Way^. Vertex]);
  Draw_Vertex (Half_Way2^. Vertex,
                Graph_G^ [Half_Way2^. Vertex]);
  End_X := Graph_G^ [Half_Way^. Vertex]. X;
  Graph_G^ [Half_Way^. Vertex]. Queued := true;
  Graph_G^ [Half_Way2^. Vertex]. Queued := true;
  Start_Vertex := Saved_Start_Vertex
end;
```

```
{-------------------------------------------------------}
{- Now we proceed to draw half of the polygon between      -}
{-     Start_X and End_X, Start_Y_Coord and Boundary_Y      -}
{- This half is the 'bottom' half of the convex polygon.    -}
{- There are Number points in this half.                    -}
{-------------------------------------------------------}
procedure Draw_Half_Poly (Number, Start_Y_Coord, Boundary_Y : Coords);

var
  Loop,
  Number_per_Side : integer;
  Current_Y_Inc,
  Total_Y_Inc,
  X_Inc, Y_Inc,
  X_Pos, Y_Pos : real;

begin
  X_Inc := (End_X - Start_X) / (Number + 1);     { regular intervals on X-axis }
  X_Pos := Start_X;
  Y_Pos := Start_Y_Coord;
  Total_Y_Inc := (Boundary_Y - Offset - Start_Y_Coord);   { calculate total y increment }
  Number_Per_Side := (Number div 2) + (Number mod 2);
                                                          { simple summation formula yields }
  Y_Inc := (2.0 * Total_Y_Inc) /                          { the unit y increment            }
              ((sqr(1.0 * Number_per_Side) + (Number_per_Side)));

  Current_Y_Inc := Y_Inc * (Number_per_Side);             { large initial y increment }
  For Loop := 1 to (Number div 2) do                      { which will decrease regularly to }
    begin                                                 { ensure interior angles less than 180 }
      X_Pos := X_Pos + X_Inc;
      Y_Pos := Y_Pos + Current_Y_Inc;         { compute new coordinate }
      Current_Y_Inc := Current_Y_Inc - Y_Inc;             { decrease next increment }
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);  { save in for point }
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,                    { and draw the vertex }
                   Graph_G^ [Start_Vertex^. Vertex]);
      Get_Next_Element (Start_Vertex)
    end;
  if Number mod 2 = 1              { special case for odd number per side }
    then begin                     { simply do an extra iteration }
      X_Pos := X_Pos + X_Inc;
      Y_Pos := Y_Pos + Current_Y_Inc;
      Current_Y_Inc := Current_Y_Inc - Y_Inc;
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex]);
      Get_Next_Element (Start_Vertex);
      Current_Y_Inc := Y_Inc;
    end;
  For Loop := 1 to (Number div 2) do          { now we have reached our midpoint, }
    begin                                     { so we start to come back to centre }
      X_Pos := X_Pos + X_Inc;                 { of screen again and increase gradient }
      Y_Pos := Y_Pos - Current_Y_Inc;
      Current_Y_Inc := Current_Y_Inc + Y_Inc;  { add back what we have been taking off }
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);
```

```
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex]);  { and place each vertex }
      Get_Next_Element (Start_Vertex)
    end
end;

{-------------------------------------------------------}
{- Now we proceed to draw top half of the polygon between   -}
{-     Start_X and End_X, Start_Y_Coord and Boundary_Y      -}
{- There are Number points in this half.                    -}
{-------------------------------------------------------}
procedure Draw_Other_Half_Poly (Number, Start_Y_Coord, Boundary_Y : Coords);

var
  Loop,
  Number_per_Side : integer;
  Current_Y_Inc,
  Total_Y_Inc,
  X_Inc, Y_Inc,
  X_Pos, Y_Pos : real;

begin
  Get_First_Element (Initial_Cycle, Start_Vertex);
  Get_Prev_Element (Start_Vertex);
  X_Inc := (End_X - Start_X) / (Number + 1);
  X_Pos := Start_X;
  Y_Pos := Start_Y_Coord;
                            { same calculations as in previous procedure }

  Total_Y_Inc := (Boundary_Y + Offset - Start_Y_Coord);
  Number_Per_Side := (Number div 2) + (Number mod 2);
  Y_Inc := (2.0 * Total_Y_Inc) / ((sqr(1.0 * Number_per_Side) + (Number_per_Side)));
  Current_Y_Inc := Y_Inc * (Number_per_Side);

  For Loop := 1 to (Number div 2) do     { place last half of vertices first }
    begin                                { we work back from right to left this time }
      X_Pos := X_Pos + X_Inc;
      Y_Pos := Y_Pos + Current_Y_Inc;
      Current_Y_Inc := Current_Y_Inc - Y_Inc;    { but same principle of gradients apply }
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex]);   { place each vertex }
      Get_Prev_Element (Start_Vertex)
    end;
  if Number mod 2 = 1              { again a special case for the midpoint }
    then begin
      X_Pos := X_Pos + X_Inc;
      Y_Pos := Y_Pos + Current_Y_Inc;
      Current_Y_Inc := Current_Y_Inc - Y_Inc;
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex]);
```

```
      Get_Prev_Element (Start_Vertex);
      Current_Y_Inc := Y_Inc;
    end;
  For Loop := 1 to (Number div 2) do    { and we place the latter half by steadily }
    begin                               { increasing the gradient }
      X_Pos := X_Pos + X_Inc;
      Y_Pos := Y_Pos - Current_Y_Inc;
      Current_Y_Inc := Current_Y_Inc + Y_Inc;
      Graph_G^ [Start_Vertex^. Vertex]. X := Round (X_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Y := Round (Y_Pos);
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex]);
      Get_Prev_Element (Start_Vertex)
    end
end;

{-----------------------------------------------------------}
{ The main procedure to draw the exterior polygon.          -}
{-----------------------------------------------------------}
begin
  Offset := (Max_Y - Min_Y) div Initial_Cycle. Length;
  Place_Two_Vertices;               { place first two vertices }
  if Initial_Cycle. Length = 3      { only three vertices is a special case - triangle }
    then begin
      Get_First_Element (Initial_Cycle, First);
      Second := First;
      Get_Next_Element (Second);                              { place the third vertex }
      Graph_G^ [Start_Vertex^. Vertex]. Y :=
                     (Graph_G^ [First^. Vertex]. Y +
                      Graph_G^ [Second^. Vertex]. Y) div 2;
      Graph_G^ [Start_Vertex^. Vertex]. X := Max_X - Offset;
      Graph_G^ [Start_Vertex^. Vertex]. Queued := true;
      Draw_Vertex (Start_Vertex^. Vertex,
                   Graph_G^ [Start_Vertex^. Vertex])          { and draw it }
    end
    else begin
      Place_Another_Two_Vertices;                             { place two vertices on RHS of screen }
        { draw half polygon between Start_X and End_X }
        { First_Y and (Max_Y - Offset)                 }
      if Initial_Cycle. Length > 4         { if 4 then cycle already drawn }
        then begin
          if Initial_Cycle. Length > 5
            then Draw_Half_Poly (Floor (Initial_Cycle. Length - 4),   { complete drawing }
                                 First_Y, Max_Y);

        { draw half polygon between Start_X and End_X }
        { Second_Y and (Min_Y + Offset)                }

          Draw_Other_Half_Poly (Ceiling (Initial_Cycle. Length - 4),  { complete drawing }
                                Second_Y, Min_Y);
        end
    end
end;
```

```
{-------------------------------------------------------}
{- This procedure will draw a cycle S* with S already drawn -}
{- Concepts rely heavily on Section 3.4                    -}
{-------------------------------------------------------}
procedure Draw_a_Cycle (V1, V2, A1 : Vertex;
                        A_Cycle    : Cycle);

var
  Current : Cycle_Element_Ptr;  { the current vertex we are drawing }
  Swopped : Boolean;            { if we have swopped axis or not }
  Up_X,
  Theta   : real;               { angle between the base vertices }

{-------------------------------------------------------}
{- Translate the coordinates to our system                 -}
{-------------------------------------------------------}
procedure Translate_Vertex (var Vert : Vertex);

var
  Temp_X : Coords;

begin
  Temp_X := X_Trans (Theta, Vert. X, Vert. Y);     { just call our translation routines }
  Vert. Y := Y_Trans (Theta, Vert. X, Vert. Y);
  Vert. X := Temp_X
end;

{-------------------------------------------------------}
{- As in Section 3.4, we move the third apex of our triangle-}
{- to lie between the bottom two --->                      -}
{- i.e. we compute New_Peak. y                             -}
{-------------------------------------------------------}
procedure Adjust_Apex;

var
  Use      : Vertex;          { the vertex that is closer to A1 }
  m, c     : real;            { for the equation of a line }
  Old_X,
  Delta    : integer;         { change in X values along the bottom }
  Adjusted : Boolean;         { if we have changed A1 }

begin
  Delta := V2. X - V1. X;
  Old_X := A1. X;
  Adjusted := true;
  Use := V2;                   { assume that A1 is closer to V2 - so line from V2 to A1 is of interest }

                               {------------ See Section 3.4 -------------}

  if (Delta > 0)                                        { if v2 is to the right of v1 }
    then if (A1. X < V1. X)
           then A1. X := V1. X + Round (Delta * 0.2)    { place 20% along the way }
           else if (A1. X > V2. X)
                  then begin
                    Use := V1;                          { note that A1 is closer to V2 }
                    A1. X := V1. X + Round (Delta * 0.8)   { otherwise place 80% along the way }
                  end
                  else Adjusted := false                { otherwise no change }
```

```
    else if (A1. X > V1. X)
           then A1. X := V1. X + Round (Delta * 0.2)
           else if (A1. X < V2. X)
                  then begin
                    A1. X := V1. X + Round (Delta * 0.8);
                    Use := V1
                  end
                  else Adjusted := false;
  if Adjusted                              { see if we have changed A1. x }
    then begin
      m := (A1. Y - Use. Y) / (Old_X - Use. X);    { compute the valid y coord for A1 }
      c := Use. Y - m * Use. X;
      A1. Y := Round (m * A1. X + c);
    end
end;

{-------------------------------------------------------}
{- We draw an arc of Num_pts                           -}
{-------------------------------------------------------}
procedure Draw_Arc (Num_Pts : Integer;
                    Dir : Boolean;                    { are we going up to halfway or down from halfway }
                    Org_Vert1, Org_Vert2 : Vertex;    { the base vertices of the triangle }
                    Saved_Half_Way : Vertex_Ptr_Range); { the halfway starting point }

const
  Start_Factor = 0.99;

var
  Loop          : Vertex_Ptr_Range;
  X_Pos, Y_Pos,
  Sum,
  Factor,
  Local_Theta,
  Total_Y_Inc,
  Start_Y_Inc,
  Y_Increment,
  X_Increment   : real;
  Num_per_Side  : integer;
  Vert1, Vert2,
  temp          : vertex;

{-------------------------------------------------------}
{- We use Bisection method to find the correct fact Factor  -}
{- with which we may decrease the gradient Start with in    -}
{- order to reach a total of Global in k decreases.         -}
{-                                                          -}
{- Please see Section 3.4                                   -}
{-------------------------------------------------------}
procedure Solve_Numerically (Global, Start, k : real; var Factor : real);

const
  epsilon = 0.0001;            { good enough error! }

{-------------------------------------------------------}
{- See Section 3.4 - this is the function to evaluate the   -}
{- correct factor to decrease the gradient by.              -}
{-------------------------------------------------------}
function f(x : real) : real;
```

```
begin
  f := Global - Start * ((1 - exp ((Num_per_Side) * ln (x))) / (1 - x))
end;

var
  Left,              { left value of the bisection interval }
  Right,             { right value of the bisection interval }
  New_Guess,         { halfway between the two interval endpoints }
  New_Value : real;  { and the value f(New_Guess) }

begin
  left := 0;         { positive }
  right := 0.99;     { negative }

  repeat                                    { standard bisection algorithm }
    new_guess := (left + right) / 2;
    new_value := f(new_guess);
    if new_value < 0
      then right := new_value
      else left := new_value
  until abs (left - right) < epsilon;
  Factor := left                            { return the value }
end;

begin
  Vert1 := Org_Vert1;        { temporary values }
  Vert2 := Org_Vert2;
  Local_Theta := Calculate_Theta (Vert1, Vert2);    { calculate theta relative to our system }
  Num_per_Side := (Num_Pts div 2);                  { number of points on either side }

  X_Increment := (Vert2. X - Vert1. X) / (Num_per_Side + 1);    { spaced out evenly according to X }

  if Dir = Down         { if we are proceeding from halfway back down to base }
    then begin
      Start_Y_Inc := 0.6 * X_Increment * sin (Local_Theta) / cos (Local_Theta);
      Solve_Numerically (Abs(Global_Y), Abs(Start_Y_Inc),     { find correct Factor for gradient }
                         Num_per_Side, Factor);
      Y_Increment := Start_Y_Inc
    end
    else begin
      Start_Y_Inc := 0.6 * X_Increment * sin (Local_Theta) / cos (Local_Theta);
      Total_Y_Inc := (Vert2. Y - Vert1. Y);              { find total allowed inc }
      Total_Y_Inc := Total_Y_Inc * 0.7;                  { reduce by an aesthetic factor }
      Factor := Start_Factor;                            { and find correct decrement of gradient }
      repeat
        Sum := (1 - exp ((Num_per_Side + 1) * ln (Factor))) / (1 - Factor);
        if Abs (Sum * Start_Y_Inc) > Abs (Total_Y_Inc)
          then Factor := Factor * 0.9
      until Abs (Sum * Start_Y_Inc) < Abs (Total_Y_Inc);  { until decrement is sufficient }
      Y_Increment := Start_Y_Inc
    end;

  X_Pos := Vert1. X;                    { now we may place the vertices }
  Y_Pos := Vert1. Y;
  Sum := 0;
  If Num_per_Side <> 0                  { see if any to do }
    then begin
```

```
      For Loop := 1 to Num_Pts div 2 do
        begin
          Sum := Sum + Y_Increment;        { move along in evenly spaced intervals }
          X_Pos := X_Pos + X_Increment;
          Y_Pos := Y_Pos + Y_Increment;
          Y_Increment := Y_Increment * Factor;   { and slowly decrease gradient }
          if Swopped
            then begin
              Graph_G^ [Current^. Vertex]. X :=
                        Reverse_Y_Trans (Theta, X_Pos, Y_Pos);
              Graph_G^ [Current^. Vertex]. Y :=
                        Reverse_X_Trans (Theta, X_Pos, Y_Pos)
            end
            else begin
              Graph_G^ [Current^. Vertex]. X :=
                        Reverse_X_Trans (Theta, X_Pos, Y_Pos);
              Graph_G^ [Current^. Vertex]. Y :=
                        Reverse_Y_Trans (Theta, X_Pos, Y_Pos)
            end;
          Draw_Vertex (Current^. Vertex,
                       Graph_G^ [Current^. Vertex]);  { the vertex is placed }

          if Dir = Up                                 { and get succeeding element }
            then Get_Next_Element (Current)
            else Get_Prev_Element (Current)
        end
    end;

  if Dir = Up
    then begin
      Global_Y := Sum;       { remember the halfway stage values }
      Up_X := X_Pos;
      Saved_Half_Way := Current^. Vertex;
      Current := a_Cycle. Head^. Prev;                    { and work backwards from end of cycle }
      Draw_Arc (Num_pts, Down, V2, A1, Saved_Half_Way)    { draw the rest of the cycle }
    end
    else if Num_Pts mod 2 <> 0          { lastly place the midpoint vertex }
      then begin
        X_Pos := X_Pos + (Up_X - X_Pos) / 2;
        Y_Pos := Y_Pos + Y_Increment / 2;
        if Swopped
          then begin
            Graph_G^ [Saved_Half_Way]. X :=
                      Reverse_Y_Trans (Theta, X_Pos, Y_Pos);
            Graph_G^ [Saved_Half_Way]. Y :=
                      Reverse_X_Trans (Theta, X_Pos, Y_Pos)
          end
          else begin
            Graph_G^ [Saved_Half_Way]. X :=
                      Reverse_X_Trans (Theta, X_Pos, Y_Pos);
            Graph_G^ [Saved_Half_Way]. Y :=
                      Reverse_Y_Trans (Theta, X_Pos, Y_Pos)
          end;
        Draw_Vertex (Saved_Half_Way,
                     Graph_G^ [Saved_Half_Way])
      end
end;
```

```
{-------------------------------------------------------}
{- Special case - there is only one new element to draw.    -}
{-------------------------------------------------------}
procedure Special_Case;

var
  X_Pos,
  Y_Pos,
  m, c    : real;   { for the equation of the line }

begin
   Y_Pos := (A1. Y + V1. Y) / 2;                          { get mid point }
   m := (A1. Y - V1. Y) / (A1. X - (V2. X + V1. X) / 2); { gradient }
   if m = 0
     then X_Pos := (V2. X + V1. X) / 2                    { check if misbehaved }
     else begin
       c := A1. Y - m * A1. X;                            { get intercept }
       X_Pos := (Y_Pos - c) / m                           { get x position }
     end;
   Get_First_Element (a_Cycle, Current);                  { get the only element }
   if Swopped                                             { and assign the x and y positions correctly }
     then begin
       Graph_G^ [Current^. Vertex]. X :=
                  Reverse_Y_Trans (Theta, X_Pos, Y_Pos);
       Graph_G^ [Current^. Vertex]. Y :=
                  Reverse_X_Trans (Theta, X_Pos, Y_Pos)
     end
     else begin
       Graph_G^ [Current^. Vertex]. X :=
                  Reverse_X_Trans (Theta, X_Pos, Y_Pos);
       Graph_G^ [Current^. Vertex]. Y :=
                  Reverse_Y_Trans (Theta, X_Pos, Y_Pos)
     end;
   Draw_Vertex (Current^. Vertex,
               Graph_G^ [Current^. Vertex])                { finally place the vertex }
end;
```

```
{-------------------------------------------------------}
{- The main drawing procedure to draw the extended cycle.   -}
{-------------------------------------------------------}
begin
  if Abs (V1. X - V2. X) < Abs (V1. Y - V2. Y)     { find optimal system of axis }
    then begin
      Swop (V1. X, V1. Y);    { exchange x and y for all the apexes }
      Swop (V2. X, V2. Y);
      Swop (A1. X, A1. Y);
      Swopped := true         { and note that a swop has been done }
    end
    else Swopped := false;
  Theta := Calculate_Theta (V1, V2);    { find angle between the two points }
  Translate_Vertex (A1);                { get new x and y for the points }
  Translate_Vertex (V1);
  Translate_Vertex (V2);
  Adjust_Apex;                          { as per Section 3.4 get New_Peak between V1 and V2 }
  Get_First_Element (a_Cycle, Current);
  if A_Cycle. Length div 2 > 0
    then Draw_Arc (A_Cycle. Length, Up, V1, A1, 0)   { and draw the cycle }
    else Special_Case                                { only one element }
end;


{-------------------------------------------------------}
{- We add all vertices on the new cycle which must be drawn -}
{- but are not allowed to be apexes. Thus, we space them    -}
{- equally between vertices which we have placed already.   -}
{- Draw_Cycle is the cycle of Apices,                       -}
{- Temp_Cycle is the cycle of all elements.                 -}
{-------------------------------------------------------}
procedure Add_Extra_Coords (var Draw_Cycle, Temp_Cycle : Cycle;
                            Last_Vertex, First_Vertex  : Vertex_Ptr_Range);

var
  Old_Posn,
  Temp_Posn,
  Draw_Posn   : Cycle_Element_Ptr;
  Temp_Cycle2 : Cycle;
  Count,
  Divisor     : integer;
  Temp_X,
  Temp_X_Inc,
  Temp_Y,
  Temp_Y_Inc  : real;

begin                                 { add First_Vertex to front of each cycle }
                                      { and Last_Vertex to end of each cycle }
  Add_to_Cycle (Draw_Cycle, Last_Vertex);
  Init_Cycle (Temp_Cycle2);
  Add_to_Cycle (Temp_Cycle2, First_Vertex);
  Add_Cycles (Temp_Cycle2, Draw_Cycle);
  Draw_Cycle := Temp_Cycle2;
  Init_Cycle (Temp_Cycle2);
  Add_to_Cycle (Temp_Cycle2, First_Vertex);
  Add_Cycles (Temp_Cycle2, Temp_Cycle);
  Temp_Cycle := Temp_Cycle2;
  Add_to_Cycle (Temp_Cycle, Last_Vertex);
```

```
                                        { we now guarantee that the cycles have known apexes }

  Get_First_Element (Temp_Cycle, Temp_Posn);    { start at front of each list }
  Get_First_Element (Draw_Cycle, Draw_Posn);
  while (Temp_Posn^. Next <> Temp_Cycle. Head) do   { whilst not at end of doubly linked list }
    begin
      if (Temp_Posn^. Vertex = Draw_Posn^. Vertex)  { two elements are equal so must be placed already }
        then begin
          Get_Next_Element (Temp_Posn);
                  Get_Next_Element (Draw_Posn)
        end
        else begin                                  { we must place some vertices }
          Old_Posn := Temp_Posn;
          Get_Prev_Element (Old_Posn);              { go back to known element }
          Count := 0;
          repeat
            Get_Next_Element (Temp_Posn);
            Count := Count + 1
          until (Temp_Posn^. Vertex = Draw_Posn^. Vertex);   { and find next known element }
          Temp_X := (Graph_G^ [Draw_Posn^. Vertex]. X -
                      Graph_G^ [Old_Posn^. Vertex]. X);
          Temp_Y := (Graph_G^ [Draw_Posn^. Vertex]. Y -
                      Graph_G^ [Old_Posn^. Vertex]. Y);
          Divisor := Count + 1;
          Temp_X_Inc := Temp_X / Divisor;              { calculate spacing along the line }
          Temp_Y_Inc := Temp_Y / Divisor;
          Temp_X := (Graph_G^ [Old_Posn^. Vertex]. X);  { and start position }
          Temp_Y := (Graph_G^ [Old_Posn^. Vertex]. Y);
          Get_Next_Element (Old_Posn);
          While Old_Posn <> Temp_Posn do       { place the extra vertices }
            begin
              Graph_G^ [Old_Posn^. Vertex]. X :=
                                        Round (Temp_X + Temp_X_Inc);
              Graph_G^ [Old_Posn^. Vertex]. Y :=
                                        Round (Temp_Y + Temp_Y_Inc);
              Graph_G^ [Old_Posn^. Vertex]. Queued := true;
              Draw_Vertex (Old_Posn^. Vertex,
                           Graph_G^ [Old_Posn^. Vertex]);    { and draw it! }
              Get_Next_Element (Old_Posn);
              Temp_X := Temp_X + Temp_X_Inc;
              Temp_Y := Temp_Y + Temp_Y_Inc
            end
        end
    end
end;
```

```
{-------------------------------------------------------}
{- The initial cycle is extended as per Section 3.4 for a   -}
{- block of  G - v.                                        -}
{-------------------------------------------------------}
procedure Extend_Convex_Drawing (Initial_Cycle : Cycle);

var
  First_Vertex   : Boolean;      { is this the first vertex in the new cycle? }
  Start_Vertex,
  Current_Vertex : Vertex_Ptr;   { temporary vars for the traversal of a face }
  Saved_AntiClock,
  Travel_Edge,
  Temp_Edge      : Edge_Ptr;     { ditto }
  Draw_Cycle,                    { cycle of new apexes that we have placed }
  Temp_Cycle,                    { cycle of all elements on the new cycle we have placed }
  Block_Cycle    : Cycle;        { temporary cycle }
  A1             : Cycle_Element_Ptr;  { our chosen apex }
  V1,                                  { the start vertex on the known cycle }
  V2             : Vertex;             { the end vertex on the known cycle }

begin
  Get_First_Element (Initial_Cycle, A1);     { choose an arbitrary element of Initial_Cycle }
                                             { which becomes our apex }
  if Graph_G^ [A1^. Vertex]. Deleted         { if we have selected A1 as an apex before, then exit }
    then exit;

  Graph_G^ [A1^. Vertex]. Deleted := true;   { note selected as an apex }

  Temp_Edge := Graph_G^ [A1^. Vertex]. Cyc_AntiClock;  { now repeat until we process all edges }
                                                       { between anticlock and clockwise }
  repeat
    Init_Cycle (Block_Cycle);
    Init_Cycle (Draw_Cycle);
    V1 := Graph_G^ [Temp_Edge^. Vertex];               { our start vertex on the cycle }
    Start_Vertex := Temp_Edge^. Vertex;                { move along counter clockwise edge }
    repeat                                             { until at an edge incident with A1 }
      Current_Vertex := Temp_Edge^. Vertex;
      Travel_Edge := Temp_Edge^. Other_Edge;
      Temp_Edge := Temp_Edge^. Next;
      First_Vertex := true;
      repeat                                           { build a sub path }
        Graph_G^ [Current_Vertex]. Cyc_ClockWise := Travel_Edge^. Previous;
        if (not First_Vertex)
          then begin
            Add_to_Cycle (Block_Cycle, Current_Vertex);        { only add to cycle if internal vertex }
            Graph_G^ [Current_Vertex]. Queued := true
          end
          else First_Vertex := false;
        Current_Vertex := Travel_Edge^. Previous^. Vertex;     { now proceed along a face }
        Travel_Edge := Travel_Edge^. Previous^. Other_Edge;
        Graph_G^ [Current_Vertex]. Cyc_AntiClock := Travel_Edge;
        Saved_AntiClock := Graph_G^ [Current_Vertex]. Cyc_AntiClock;
      until (A1^. Vertex = Travel_Edge^. Previous^. Vertex) or
            (Graph_G^ [Current_Vertex]. Placed);

      if (not Graph_G^ [Current_Vertex]. Queued)       { check if we must add to cycle }
        then begin
          Add_to_Cycle (Draw_Cycle, Current_Vertex);
```

```
        Add_to_Cycle (Block_Cycle, Current_Vertex);
        Graph_G^ [Current_Vertex]. Queued := true
      end
  until (Graph_G^ [Current_Vertex]. Placed);          { until we are back at a placed vertex }

  V2 := Graph_G^ [Current_Vertex];                    { becomes third part of triangle in Section 3.4 }

  if Draw_Cycle. Length > 0
    then Draw_a_Cycle (V1, V2, Graph_G^ [A1^. Vertex], Draw_Cycle);   { see if anything to draw }
  if Block_Cycle. Length > Draw_Cycle. Length
    then Add_Extra_Coords (Draw_Cycle, Block_Cycle,   { and place any extras on straight lines }
                           Current_Vertex, Start_Vertex);

  Add_to_Cycle (Block_Cycle, Current_Vertex);         { add last vertex to current queue }

  if not (Graph_G^ [V2. Number]. Deleted)
     and not (Graph_G^ [V1. Number]. Deleted)
       then Extend_Convex_Drawing (Block_Cycle);      { and recurse to draw new cycle }

  Graph_G^ [Current_Vertex]. Cyc_Clockwise :=
                Graph_G^ [Current_Vertex]. Cyc_AntiClock^. Previous;   { modify after recursion }
                                                                       { since cycle is now smaller }
  Graph_G^ [Current_Vertex]. Cyc_AntiClock := Saved_AntiClock;         { and restore modified }
until (Temp_Edge = Graph_G^ [A1^. Vertex]. Cyc_Clockwise^. Previous)
end;

{-------------------------------------------------------}
{- We add the edges into the drawing. Easy routine after the-}
{- vertices have been placed.                            -}
{-------------------------------------------------------}
procedure Add_Edges_into_Drawing;

var
  Loop      : Vertex_Ptr;
  Temp_Edge : Edge_Ptr;

begin
  For Loop := 1 to Last_Vertex do            { for every vertex }
    if Graph_G^ [Loop]. Placed then begin      { check if it was not deleted }
      Temp_Edge := Graph_G^ [Loop]. Edges;     { and draw edge to every nbour }
      repeat
        if not Temp_Edge^. Drawn               { that has no drawn edge between them already }
          then begin
            Line (Graph_G^ [Loop]. X, Graph_G^ [Loop]. Y,
                  Graph_G^ [Temp_Edge^. Vertex]. X, Graph_G^ [Temp_Edge^. Vertex]. Y);
            Temp_Edge^. Drawn := true;
            Temp_Edge^. Other_Edge^. Drawn := true   { note that the edge is placed }
          end;
        Temp_Edge := Temp_Edge^. Next
      until Temp_Edge = Graph_G^ [Loop]. Edges
    end;
end;
```

```
{-------------------------------------------------------------}
{- Perform the actual drawing of the graph.                  -}
{-------------------------------------------------------------}
procedure Convex_Drawing;

var
  Outstr : string;    { for display purposes during the demo }

begin
  Init_Graphics;
  Draw_Initial_Cycle;                       { place extendible convex cycle }
  Remove_Degree_2_Vertices;                 { remove all vertices of degree 2 not on cycle }
  if Initial_Cycle. Length <> Last_Vertex
    then
      Extend_Convex_Drawing (Initial_Cycle); { and draw the rest ---> looks easy! }
  Add_Degree_2_Vertices_to_Drawing;         { add degree 2 vertices back to drawing after completion }
  Add_Edges_into_Drawing;                   { and finally draw the edges }
  SetColor(Red);
  if Do_Demo                                { if demo then bells and whistles }
    then begin
      str(Filenumber-1, Outstr);
      SetTextStyle (DefaultFont, Horizdir, 2);
      OutTextXY(GetMaxX - 130, 10, 'Graph '+Outstr)
    end;
  SetTextStyle (DefaultFont, Horizdir, 1);
  OutTextXY(GetMaxX - 210, GetMaxY - 20, 'Press Any Key to Continue');
  GR_Prompt;           { wait for user to press a key }
  CloseGraph           { finish graphics mode, return to text }
end;

end.
```

```
{-------------------------------------------------------}
{- This unit contains the global routines and definitions  -}
{- used by all the drawing units.                          -}
{-------------------------------------------------------}
unit Convex_Globals;

interface

uses
  Planar_Defs;

type
  Two_Degree_List_Ptr  = ^Two_Degree_List_Type;      { a list of vertices of degree 2 are kept }
  Two_Degree_List_Type = record
                           Vertex : Vertex_Ptr;
                           Edges  : Edge_Ptr;
                           Next   : Two_Degree_List_Ptr
                         end;

var
  Graph_Bup       : Graph_Ptr;              { a backup graph }
  Last_Vertex_Bup : Vertex_Ptr_Range;       { and last vertex backup }
  Forbidden_Pair  : Boolean;                { is there a forbidden pair }
  Two_Degree_List : Two_Degree_List_Ptr;    { and the list of vertices of degree 2 }

procedure Init_Cycle (var a_Cycle : Cycle);            { initialise a cycle }

procedure Add_to_Cycle (var a_Cycle : Cycle;
                        Current_Vertex : Vertex_Ptr);  { add an element to a cycle }

procedure Add_Cycles (var Cycle1, Cycle2 : Cycle);     { add two cycles together }

procedure Dump_Cycle (a_Cycle : Cycle);                { display a cycle on screen }

procedure Get_First_Element (a_Cycle : Cycle;
                             var Current : Cycle_Element_Ptr);   { get first element of a cycle }

procedure Get_Next_Element (var Current : Cycle_Element_Ptr);    { get next element of a cycle }

procedure Get_Prev_Element (var Current : Cycle_Element_Ptr);    { get previous element of a cycle }

procedure Reverse_Cycle (var a_Cycle : Cycle);          { reverse a cycle }

procedure Restore_Graph;                   { restore a saved backup of a graph }

procedure Remove_Degree_2_Vertices;        { as the name suggests! }

procedure Add_Degree_2_Vertices_to_Drawing; { as the name suggests }

procedure Draw_Vertex (a_Number : integer;  { draw a vertex on the screen with a number }
                       var a_Vert : Vertex);


implementation

uses
  Graph,
  Planar_Miscellaneous;
```

```
{-------------------------------------------------------------}
{- Initialise Cycle routine - only one of two routines to   -}
{- know the data structure of a cycle.                      -}
{-------------------------------------------------------------}
procedure Init_Cycle (var a_Cycle : Cycle);

begin
  a_Cycle. Length := 0;     { merely note length is zero }
  a_Cycle. Head := nil;
end;

{-------------------------------------------------------------}
{- A vertex is added to the cycle.                          -}
{-------------------------------------------------------------}
procedure Add_to_Cycle (var a_Cycle : Cycle;
                        Current_Vertex : Vertex_Ptr);
var
  Temp_Element : Cycle_Element_Ptr;

begin
  a_Cycle. Length := a_Cycle. Length + 1;   { increase length }
  new (Temp_Element);
  Temp_Element^. Vertex := Current_Vertex;            { and standard insertion into list }
  if a_Cycle. Head <> nil
    then begin
      Temp_Element^. Prev := a_Cycle. Head^. Prev;
      a_Cycle. Head^. Prev^. Next := Temp_Element;
      a_Cycle. Head^. Prev := Temp_Element;
      Temp_Element^. Next := a_Cycle. Head
    end
    else begin
      a_Cycle. Head := Temp_Element;
      Temp_Element^. Next := Temp_Element;
      Temp_Element^. Prev := Temp_Element
    end
end;
```

```
{-------------------------------------------------------}
{- Add two cycles together                              -}
{-------------------------------------------------------}
procedure Add_Cycles (var Cycle1, Cycle2 : Cycle);

var
  First1, First2,
  Last1, Last2    : Cycle_Element_Ptr;

begin
  if Cycle2. Length > 0
    then
      if Cycle1. Head = Nil
        then Cycle1 := Cycle2
        else begin
          First1 := Cycle1. Head;            { merely concatenate the two lists }
          First2 := Cycle2. Head;
          Last1 := Cycle1. Head^. Prev;
          Last2 := Cycle2. Head^. Prev;
          First1^. Prev := Last2;
          Last2^. Next := First1;
          Last1^. Next := First2;
          First2^. Prev := Last1;
          Cycle1. Length := Cycle2. Length + Cycle1. Length
        end
end;

{-------------------------------------------------------}
{- By traversing a doubly linked list, we display a cycle   -}
{-------------------------------------------------------}
procedure Dump_Cycle (a_Cycle : Cycle);

var
  Start,
  Temp : Cycle_Element_Ptr;

begin
  writeln ('The cycle is : ');
  Get_First_Element (a_Cycle, Start);
  Temp := Start;
  if Temp <> nil
    then begin
      repeat
        write (' ', Temp^.Vertex);
        Get_Next_Element (Temp)
      until Temp = Start
    end
    else writeln ('empty');
  writeln;
  prompt
end;
```

```
{-------------------------------------------------------}
{- Merely accesses the first element of a cycle.       -}
{-------------------------------------------------------}
procedure Get_First_Element (a_Cycle : Cycle;
                             var Current : Cycle_Element_Ptr);

begin
  Current := a_Cycle. Head
end;

{-------------------------------------------------------}
{- Gets the next element in doubly linked list for cycles.  -}
{-------------------------------------------------------}
procedure Get_Next_Element (var Current : Cycle_Element_Ptr);

begin
  Current := Current^. Next
end;

{-------------------------------------------------------}
{- Gets the previous element in doubly link list for cycles.-}
{-------------------------------------------------------}
procedure Get_Prev_Element (var Current : Cycle_Element_Ptr);

begin
  Current := Current^. Prev
end;

{-------------------------------------------------------}
{- A cycle is reversed!!.                              -}
{-------------------------------------------------------}
procedure Reverse_Cycle (var a_Cycle : Cycle);

var
  Current,
  Saved_Start,
  Next,
  Prev        : Cycle_Element_Ptr;

begin
  Current := a_Cycle. Head;
  Saved_Start := Current;
  Prev := a_Cycle. Head^. Prev;
  a_Cycle. Head := Prev;           { merely traverse the list swopping pointers }
  repeat
    Next := Current^. Next;
    Current^. Next := Prev;
    Current^. Prev := Next;
    Prev := Current;
    Current := Next;
  until (Current = Saved_Start);
end;
```

```
{-------------------------------------------------------}
{- Restore a saved graph                                -}
{- Note that we keep the calculated X and Y coordinates.    -}
{-------------------------------------------------------}
procedure Restore_Graph;

var
  Loop : Vertex_Ptr;

begin
  For Loop := 1 to Last_Vertex do   { first replicate the X and Y coords }
    begin
      Graph_Bup^ [Loop]. X := Graph_G^ [Loop]. X;
      Graph_Bup^ [Loop]. Y := Graph_G^ [Loop]. Y
    end;
  Scratch_Graph;
  Last_Vertex := Last_Vertex_Bup;   { then restore the graph }
  Graph_G^ := Graph_Bup^;
end;

{-------------------------------------------------------}
{- Remove degree 2 vertices and place on a list.           -}
{-------------------------------------------------------}
procedure Remove_Degree_2_Vertices;

var
  Temp_Degree_List : Two_Degree_List_Ptr;
  Nbour1, NBour2,
  Loop             : Vertex_Ptr;
  Temp_Edge2,
  Temp_Edge        : Edge_Ptr;

begin
  Two_Degree_List := nil;
  Num_Deleted := 0;
  For Loop := 1 to Last_Vertex do
    if (Graph_G^ [Loop]. Edges^. Next^. Next = nil)         { must be degree 2 }
      and (not Graph_G^ [Loop]. Placed)
      then begin
        Num_Deleted := Num_Deleted + 1;
        Temp_Edge := Graph_G^ [Loop]. Edges;
        writeln ('Removing Vertex ', Loop, ' temporarily... nbours are : ',
                 Temp_Edge^. Vertex, ' and ', temp_Edge^. Next^. Vertex);
        Nbour1 := Temp_Edge^. Vertex;
        Nbour2 := Temp_Edge^. Next^. Vertex;
        Temp_Edge2 := Graph_G^ [Nbour1]. Edges;
        While (Temp_Edge2 <> Nil) and (Temp_Edge2^. Vertex <> NBour2) do
          Temp_Edge2 := Temp_Edge2^. Next;
        if Temp_Edge2 = nil
          then begin
            Temp_Edge^. Other_Edge^. Vertex := Nbour2;
            Temp_Edge^. Other_Edge^. Other_Edge := Temp_Edge^. Next^. Other_Edge;
            Temp_Edge^. Next^. Other_Edge^. Other_Edge := Temp_Edge^. Other_Edge;
            Temp_Edge^. Next^. Other_Edge^. Vertex := Nbour1;
            new (Temp_Degree_List);
            Temp_Degree_List^. Next := Two_Degree_List;
            Two_Degree_List := Temp_Degree_List;
            Temp_Degree_List^. Vertex := Loop;
```

```
            Temp_Degree_List^. Edges := Temp_Edge;
            Graph_G^ [Loop]. Deleted := true;
            Graph_G^ [Loop]. Edges := nil
          end
          else begin
            Forbidden_Pair := true;
            halt
          end
      end
end;

{-------------------------------------------------------------}
{- Draw a vertex and associated number on the screen.       -}
{-------------------------------------------------------------}
procedure Draw_Vertex (a_Number : integer;
                       var a_Vert : Vertex);

const
  Char_Offset = 5;
  X_Sign = -3;
  Y_Sign = -3;

var
  Val_Str  : String;

begin
  with a_Vert do
    begin
      Placed := true;
      if a_Number < 30
        then begin
          if FileNumber < MaxDemoFiles+1
            then begin
              Circle (X, Y, 4);
              FloodFill (X, Y, GetMaxColor)
            end
            else begin
              Circle (X, Y, 3);
              FloodFill (X, Y, GetMaxColor)
            end;
          if (not Do_Demo) or
             ((FileNumber < MaxDemoFiles+1) or ((a_Number <> 11)
                  and (a_Number <> 17) and (a_Number <> 19)
                  and (a_Number <> 20) and (a_Number <> 12)
                  and (a_Number <> 21) ))
            then begin
              str (a_Number, Val_Str);
              setcolor (Yellow);
              SetTextStyle (DefaultFont, HorizDir, 1);
              OutTextXY (X + X_Sign * Char_Offset,
                         Y + Y_Sign * Char_Offset,
                         Val_Str)
            end;
          Placed := true;
          Queued := true;
          SetColor (White);
          SetLineStyle (SolidLn, 0, 2)
        end
```

```
    end
end;

{-------------------------------------------------------------}
{- Add the vertices from the list to the graph.              -}
{-------------------------------------------------------------}
procedure Add_Degree_2_Vertices_to_Drawing;

var
  Temp_Degree_List2,
  Temp_Degree_List : Two_Degree_List_Ptr;
  Temp_Edge        : Edge_Ptr;

begin
  Temp_Degree_List := Two_Degree_List;
  while Temp_Degree_List <> nil do     { for every vertex of degree 2 }
    begin
      Graph_G^ [Temp_Degree_List^. Vertex]. Edges := Temp_Degree_List^. Edges;  { add back to graph }
      Temp_Edge := Temp_Degree_List^. Edges;
      Temp_Edge^. Other_Edge^. Vertex := Temp_Degree_List^. Vertex;
      Temp_Edge^. Next^. Other_Edge^. Vertex := Temp_Degree_List^. Vertex;
      with Temp_Edge^ do                                          { calculate the coords }
        begin
          Graph_G^ [Temp_Degree_List^. Vertex]. X :=
               (Graph_G^ [Vertex]. X + Graph_G^ [Next^. Vertex]. X) div 2;
          Graph_G^ [Temp_Degree_List^. Vertex]. Y :=
               (Graph_G^ [Vertex]. Y + Graph_G^ [Next^. Vertex]. Y) div 2
        end;
      Draw_Vertex (Temp_Degree_List^. Vertex, Graph_G^ [Temp_Degree_List^. Vertex]); { and place the vertex
      Temp_Edge^. Next^. Next := Graph_G^ [Temp_Degree_List^. Vertex]. Edges;
      Temp_Degree_List2 := Temp_Degree_List;
      Temp_Degree_List := Temp_Degree_List^. Next;
    end
end;


end.
```

```
{-------------------------------------------------------}
{- The Lempel, Even and Cederbaum Algorithm             -}
{-------------------------------------------------------}
Unit Embedd_Algorithm;

interface

procedure Lempel_Even_CederBaum_Embedding;

implementation

uses
  Planar_Defs,
  Planar_Miscellaneous,
  Embedd_Globals,          { Global Routines used by Reduction and Bubble phases }
  Embedd_Reduce,           { Reduction unit                                      }
  Embedd_Bubble,           { Bubble unit                                         }
  CRT;

{-------------------------------------------------------}
{- Main program                                         -}
{-------------------------------------------------------}
procedure Lempel_Even_Cederbaum_Embedding;

{-------------------------------------------------------}
{- The Depth-First routine is as per Hopcroft and Tarjan, -}
{-                 see Chapter 1, and Section 2.2.      -}
{- but we do not need to compute L2.                    -}
{-------------------------------------------------------}
procedure DFS;

function Min (x, y : Integer) : Integer;

begin
  if x < y
    then Min := x
    else Min := y;
end;
```

```
var
  Current_Label : 0..MaxInt;
  Temp_Edge     : Edge_Ptr;
  Finished,
  New_Vertex    : Boolean;
  Temp_Vertex   : Vertex_Ptr;

begin
{Initialising all Fathers, Labels to 0 }
  For Temp_Vertex := 1 to Last_Vertex do
    with Graph^ [Temp_Vertex] do
      begin
        Number := 0;
        Father := 0;
        L1 := 0;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do
          begin
            Temp_Edge^. Used := False;
            Temp_Edge^. Deleted := False;
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;
  New_Vertex := true;
  Temp_Vertex := 1;
  Current_Label := 0;
  repeat
    repeat
      {2}
      if New_Vertex
        then begin
          Current_Label := Current_Label + 1;
          Graph^ [Temp_Vertex]. Number := Current_Label;
          Graph^ [Temp_Vertex]. L1 := Current_Label;
        end;
      {3}
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      While (Temp_Edge <> Nil) and ((Temp_Edge^. Used) or (Temp_Edge^. Deleted)) do
        Temp_Edge := Temp_Edge^. Next;
      if (Temp_Edge <> Nil)
        then begin
          {4}
            { Direct the Edge }
            Temp_Edge^. Used := True;
            Temp_Edge^. Other_Edge^. Deleted := true;
            with Graph^ [Temp_Edge^. Vertex] do
            if Number <> 0
              then begin { Back Edge - adjust L1 and return }
                if Number < Graph^ [Temp_Vertex]. L1
                  then begin
                    Graph^ [Temp_Vertex]. L1 := Number;
                  end
                  else;
                New_Vertex := false;
              end
              else begin
                Father := Temp_Vertex;
```

```
                Temp_Vertex := Temp_Edge^. Vertex;
                New_Vertex := true;
              end;
        end
    until (Temp_Edge = Nil);
    if Graph^ [Temp_Vertex]. Number = 1
      then
        {5} Finished := true
      else begin
        {6}
        with Graph^ [Graph^ [Temp_Vertex]. Father] do
          if Graph^ [Temp_Vertex]. L1 < L1
            then L1 := Graph^ [Temp_Vertex]. L1;
        Temp_Vertex := Graph^ [Temp_Vertex]. Father;
        New_Vertex := false;
      end
  until Finished;
end;

{-------------------------------------------------------------}
{- We reorder the lists to ensure that we always choose an  -}
{- edge with the L1 weighting first.                        -}
{- The algorithm is as per Hopcroft and Tarjan, Section 2.2 -}
{-------------------------------------------------------------}
procedure ReOrder_Lists;

{-------------------------------------------------------------}
{- Simplified weighting - only L1 necessary.                -}
{-------------------------------------------------------------}
function Phi (u, v : Vertex_Ptr) : Integer;

begin
  if Graph^ [v]. Number < Graph^ [u]. Number
    then Phi := Graph^ [v]. Number
    else Phi := Graph^ [v]. L1
end;
```

```
var
  Bucket_Array : array [1..Max_Vertices] of Bucket_Ptr;
  Temp_Bucket,
  Temp_Bucket2 : Bucket_Ptr;
  Temp_Vertex  : Integer;
  Temp_Edge    : Edge_Ptr;

begin
  For Temp_Vertex := 1 to Last_Vertex do
    Bucket_Array [Temp_Vertex] := Nil;
  For Temp_Vertex := 1 to Last_Vertex do          { for each vertex }
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;       { strip each edge }
      while (Temp_Edge <> Nil) do
        begin
          Temp_Edge^. Weight := Phi (Temp_Vertex, Temp_Edge^. Vertex);   { get weight }
          new (Temp_Bucket);
          Temp_Bucket^. Data := Temp_Edge;
          Temp_Bucket^. Next := Bucket_Array [Temp_Edge^. Weight];
          Temp_Bucket^. Vertex := Temp_Vertex;
          Bucket_Array [Temp_Edge^. Weight] := Temp_Bucket;             { and add into correct bucket }
          Temp_Edge := Temp_Edge^. Next
        end;
      Graph^ [Temp_Vertex]. Edges := Nil;
    end;

  For Temp_Vertex := Last_Vertex downto 1 do        { for each bucket ---> note only p, not 2*p+1 }
    begin
      Temp_Bucket := Bucket_Array [Temp_Vertex];     { empty the bucket }
      while Temp_Bucket <> Nil do
        begin
          Temp_Bucket^. Data^. Next := Graph^ [Temp_Bucket^. Vertex]. Edges;
          Graph^ [Temp_Bucket^. Vertex]. Edges := Temp_Bucket^. Data;  { add to correct adjacency list }
          Temp_Bucket2 := Temp_Bucket;
          Temp_Bucket := Temp_Bucket^. Next;
        end;
    end;
end;
```

```
{-------------------------------------------------------}
{- The main initialise routine for all PQ-tree ops.    -}
{-------------------------------------------------------}
procedure Global_Initialise_Lempel;

const
  New_Mark = 1;   { unused vertex mark }
  Old_Mark = 0;   { used vertex mark   }

var
  Temp_Edge : Edge_Ptr;       { temporary variable }
  Loop      : Vertex_Ptr;     { temporary variable }

begin
  DFS;                                    { for the ST numbering and for 2-Connected components }
  if not Is_2_Connected
    then Is_2_Connected := Check_2_Connected;
  if Is_2_Connected
    then begin
      ReOrder_Lists;                        { to ensure we choose correct paths }
                                            { are chosen during ST-Numbering    }
      new (Upward_Embed_Graph);
      for Loop := 1 to Last_Vertex do       { reset the vertices and }
        begin                               { edges to initial value for ST Numbering }
          Upward_Embed_Graph^ [Loop]. Edges := nil;
          Upward_Embed_Graph^ [Loop]. Reverse := false;
          Graph^ [Loop]. Mark := New_Mark;
          Graph^ [Loop]. Used := false;     { we have not tested this vertex yet }
          Temp_Edge := Graph^ [Loop]. Edges;
          while Temp_Edge <> Nil do
            begin
              Temp_Edge^. Mark := New_Mark;
              Temp_Edge := Temp_Edge^. Next;
            end;
        end;
    end
    else
end;
```

```
{-------------------------------------------------------}
{- St-Numbering for the graph is computed. See chapter 1 for-}
{- details on the algorithm.                            -}
{-------------------------------------------------------}
procedure ST_Numbering (Start_Vertex : Vertex_Ptr);

const
  New_Mark = 1;   { unused vertex mark }
  Old_Mark = 0;   { used vertex mark   }

var
  ST_Stack     : Array [1..Max_Vertices] of Vertex_Ptr; { stack of vertices to give numbers to }
  Top_of_Stack : 0..Max_Vertices;                       { top of the above stack              }
  Current_Number,                  { current ST number to assign }
  Loop         : Integer;          { temporaray variable         }
  Current_Vertex : Vertex_Ptr;     { Current vertex the algorithm is at }
  Temp_Edge      : Edge_Ptr;       { temporary variable }

{-------------------------------------------------------}
{- We are adding a path from Current vertex down the tree   -}
{- until we reach a back edge.                              -}
{- The routine is necessarily recursive, since we need to   -}
{- add the path in reverse order.                           -}
{-------------------------------------------------------}
procedure Add_Forward_Path_to_Stack (Current_Edge : Edge_Ptr);

var Temp_Edge : Edge_Ptr;

begin
  if Graph^ [Current_Edge^. Vertex]. Mark <> Old_Mark   { see if we stop yet }
    then begin
      Temp_Edge := Graph^ [Current_Edge^. Vertex]. Edges;  { get first valid edge }
      while Temp_Edge^. Deleted do
        Temp_Edge := Temp_Edge^. Next;
      Add_Forward_Path_to_Stack (Temp_Edge);              { add path from there to the path }
      Current_Edge^. Mark := Old_Mark;                    { Note that the edge is used      }
      Top_of_Stack := Top_of_Stack + 1;
      ST_Stack [Top_of_Stack] := Current_Edge^. Vertex;   { and add this vertex at the TOS  }
      Graph^ [Current_Edge^. Vertex]. Mark := Old_Mark    { Note that the vertex is stacked }
    end
    else Current_Edge^. Mark := Old_Mark                  { At the end of the path          }
end;
```

```
{-------------------------------------------------------}
{- We are adding a path from the vertex in reverse direction-}
{- back along a directed path.                          -}
{- The routine is necessarily recursive, since we need to   -}
{- add the path in reverse order.                       -}
{-------------------------------------------------------}
procedure Add_Backward_Path_to_Stack (Current_Edge : Edge_Ptr);

var Temp_Edge : Edge_Ptr;

begin
  if Graph^ [Current_Edge^. Vertex]. Mark <> Old_Mark   { see if the path is finished yet }
    then begin
      Temp_Edge := Graph^ [Current_Edge^. Vertex]. Edges;
      while (Temp_Edge^. Vertex <> Graph^ [Current_Edge^. Vertex]. Father) do
        Temp_Edge := Temp_Edge^. Next;                  { get a valid edge }
      Add_Backward_Path_to_Stack (Temp_Edge);           { add the path to the stack }
      Top_of_Stack := Top_of_Stack + 1;
      ST_Stack [Top_of_Stack] := Current_Edge^. Vertex; { add this vertex }
      Current_Edge^. Mark := Old_Mark;                  { note the edge is used }
      Graph^ [Current_Edge^. Vertex]. Mark := Old_Mark  { and the vertex is on the stack }
    end
    else Current_Edge^. Mark := Old_Mark                { at the end of the path }
end;

{-------------------------------------------------------}
{- The main ST-Numbering routine                        -}
{-------------------------------------------------------}
begin
  Top_of_Stack := 2;    { Elements 1 and n only             }
  ST_Stack [1] := Start_Vertex;               { Vertex 1 is S }
  ST_Stack [2] := Graph^ [Start_Vertex]. Edges^. Vertex; { and this is T }
  Current_Number := 1;
  Graph^ [1]. Mark := Old_Mark;
  Temp_Edge := Graph^ [Start_Vertex]. Edges;
  Graph^ [Temp_Edge^. Vertex]. Mark := Old_Mark;
  Temp_Edge^. Mark := Old_Mark;
  Temp_Edge^. Other_Edge^. Mark := Old_Mark;  { Note the edge between them is used }
  while Top_of_Stack > 0 do                   { While a number needs an element }
    begin
      if Top_of_Stack = 1      { ending condition - last vertex S is on the stack }
        then begin
          Graph^ [ST_Stack [1]]. St_Number := Current_Number;  { Assign the last ST-number }
          Graph^ [ST_Stack [1]]. Used := true;                 { Note it has been tested   }
          ST_Number_Index [Current_Number] := ST_Stack [1];    { to the last vertex        }
          Top_of_Stack := 0                                    { we are finished           }
        end
        else begin
          Current_Vertex := ST_Stack [Top_of_Stack];    { get the vertex }
          Top_of_Stack := Top_of_Stack - 1;
          Temp_Edge := Graph^ [Current_Vertex]. Edges;
          while (Temp_Edge <> Nil) and (Temp_Edge^. Mark <> New_Mark) do
            Temp_Edge := Temp_Edge^. Next;         { get first unused edge }
          if Temp_Edge = Nil                       { if none left          }
            then begin
              Graph^ [Current_Vertex]. ST_Number := Current_Number;  { don't add it back }
              Graph^ [Current_Vertex]. Used := true;                 { Note it has been tested }
              ST_Number_Index [Current_Number] := Current_Vertex;    { Note the vertex assigned this number }
```

```
            Graph^ [Current_Vertex]. Mark := Old_Mark;          { give an ST - number to the vertex    }
            Current_Number := Current_Number + 1;               { and move to next numbering available }
          end
          else
            if Graph^ [Temp_Edge^. Vertex]. Number
                 < Graph^ [Current_Vertex]. Number        { check if it is a back edge }
              then begin
                Temp_Edge^. Mark := Old_Mark;             { note the edge used }
                Top_of_Stack := Top_of_Stack + 1;         { and re-add the vertex to the stack }
                ST_Stack [Top_of_Stack] := Current_Vertex
              end
              else
                if not Temp_Edge^. Deleted                { if the edge is a tree edge }
                  then begin
                    Add_Forward_Path_to_Stack (Temp_Edge);     { add the path to the stack       }
                    Top_of_Stack := Top_of_Stack + 1;          { in order so that current vertex }
                    ST_Stack [Top_of_Stack] := Current_Vertex  { is on top of the stack          }
                  end
                  else begin                              { if the edge is a back edge }
                                                          { ending at this vertex }
                    Add_Backward_Path_to_Stack (Temp_Edge);    { add the path to the stack       }
                    Top_of_Stack := Top_of_Stack + 1;          { in order so that current vertex }
                    ST_Stack [Top_of_Stack] := Current_Vertex  { is on top of the stack          }
                  end
      end
  end;
end;

{-------------------------------------------------------------}
{- Main initialisation Routine                                -}
{-------------------------------------------------------------}
procedure Initialise_LEC (Start_Vertex : Vertex_Ptr);

var
  Loop    : Vertex_Ptr;
  Temp_PQ : PQ_Node_Ptr;

begin
  ST_Numbering (Start_Vertex);      { Generate the ST-Numbering                  }
  Used_List := Nil;                 { Used List is for Tree node reinitialisation }
                                    { during the program's running.              }
  Create_PQ_Node (Temp_PQ);         { Build the root }
  Temp_PQ^. Node_Type := P_Node;
  Temp_PQ^. Child_Count := 0;
  Temp_PQ^. List_Start := Nil;
  Start_Vertex := ST_Number_Index [1];          { Note the starting vertex }
  Add_Edges_to_Tree (Start_Vertex, Temp_PQ);    { and add the leaves       }
  Sizeof_Queue := 0;                            { Nothing on the Queue         }
  Num_Vertices_Added := 1;                      { We have only added the root }
end;
```

```
{-------------------------------------------------------------}
{- The upward embedding is extednded to an embedding of G.  -}
{-------------------------------------------------------------}
procedure Extend_Embedding;

var
  Loop : Embed_Vertex_Ptr;

{-------------------------------------------------------------}
{- We copy the adjacency list from Old_List to New_List     -}
{- We need to ensure that the order is kept the same.       -}
{-------------------------------------------------------------}
procedure Duplicate_Edge_List (var New_List_Ptr, Old_List_Ptr : Embed_Edge_Ptr);

var
  Current,
  Previous,
  Dup_Current : Embed_Edge_Ptr;

begin
  Current := Old_List_Ptr;
  Previous := Nil;              { previous entry in the list }
  if Current = Nil
    then New_List_Ptr := nil
    else begin
      repeat
        new (Dup_Current);
        Dup_Current^ := Current^;  { copy the fields }
        if Previous <> nil
          then Previous^. Next := Dup_Current   { set up link from previous to Dup_Current }
          else New_List_Ptr := Dup_Current;     { set head of list correctly }
        Previous := Dup_Current;                { note new previous }
        Current := Current^. Next
      until Current = Nil;                    { until list finished }
      Previous^. Next := nil
    end
end;
```

```
{-------------------------------------------------------}
{- We extend the embedding in a DFS manner -           -}
{- as per Section 3.1                                  -}
{-------------------------------------------------------}
procedure DFS_Extend_Embed (Current_Vertex : Embed_Vertex_Ptr);

var
  Temp_Edge,
  Temp_Edge2 : Embed_Edge_Ptr;
  Next_Vertex : Embed_Vertex_Ptr;

begin
  Upward_Embed_Graph^ [Current_Vertex]. Used := true;
  Temp_Edge := Upward_Embed_Graph^ [Current_Vertex]. Edges;
  while Temp_Edge <> nil do
    begin
      Next_Vertex := Temp_Edge^. Tail_Vertex;
      new (Temp_Edge2);
      Temp_Edge2^. Indicator := false;
      Temp_Edge2^. Tail_Vertex := Current_Vertex;
      Temp_Edge2^. Next := Embed_Graph^ [Next_Vertex]. Edges;   { add to adjacency list }
      Embed_Graph^ [Next_Vertex]. Edges := Temp_Edge2;          { as per Section 3.1 }
      if not Upward_Embed_Graph^ [Next_Vertex]. Used          { if unexplored then }
        then DFS_Extend_Embed (Next_Vertex);                  { explore it! }
      Temp_Edge := Temp_Edge^. Next
    end;
end;

{-------------------------------------------------------}
{- The main procedure Extend_Embedding                 -}
{_ Again - procedure is as per Section 3.1             -}
{-------------------------------------------------------}
begin
  new (Embed_Graph);
  Embed_Graph^ := Upward_Embed_Graph^;         { make copy of entire graph }
  For Loop := 1 to Last_Vertex do
    begin
      Upward_Embed_Graph^ [Loop]. Used := false;
      Embed_Graph^ [Loop]. Edges := nil;            { bult ignore duplicate pointers }
      Duplicate_Edge_List (Embed_Graph^ [Loop]. Edges,         { and build copy instead }
                           Upward_Embed_Graph^ [Loop]. Edges)
    end;
  DFS_Extend_Embed (ST_Number_Index [Last_Vertex - Num_Deleted])  { and extend it finally }
end;
```

```
{-------------------------------------------------------------}
{- We reverse adjacency lists if necessary.                  -}
{-------------------------------------------------------------}
procedure Correct_Adjacency_Lists;

var
  Loop          : Embed_Vertex_Ptr;
  Last_Made,
  Temp_List,
  Temp_Element,
  Temp_Element2 : Embed_Edge_Ptr;

{-------------------------------------------------------------}
{- We reverse an adjacency list.                             -}
{-------------------------------------------------------------}
procedure Reverse_List (Previous, Current : Embed_Edge_Ptr);

var
  Current_Vertex : Embed_Vertex_Ptr;
  Temp_Element   : Embed_Edge_Ptr;

begin
  if Current <> Nil
    then
      if Current^. Indicator                  { if it is an indicator }
        then begin
          if Current^. Same_As                  { note that we must reverse others }
            then Upward_Embed_Graph^ [Current^. Indicator_Vertex]. Reverse := true;
          Temp_Element := Current^. Next;
          Reverse_List (Previous, Temp_Element)  { and continue with rest of list }
        end
        else begin
          Reverse_List (Current, Current^. Next);   { do the recursion }
          Current^. Next := Previous                { and perform the reversal }
        end
    else Last_Made := Previous                   { end case is special }
end;
```

```
{-------------------------------------------------------}
{- Main procedure - Correct Adjacency Lists            -}
{-------------------------------------------------------}
begin
  new (Temp_Element);
  For Loop := Last_Vertex - Num_Deleted downto 1 do     { for each vertex }
    begin                                                 { look for reversals required }
      Current_Vertex := ST_Number_Index [Loop];
      if Upward_Embed_Graph^ [Current_Vertex]. Reverse  { if you must reverse }
        then begin
          Reverse_List (Nil, Upward_Embed_Graph^ [Current_Vertex]. Edges);  { then do it!! }
          Upward_Embed_Graph^ [Current_Vertex]. Edges := Last_Made          { and note start case }
        end
        else begin
          Temp_Element^. Next := Upward_Embed_Graph^ [Current_Vertex]. Edges;  { normal case }
          Temp_List := Temp_Element;                                          { so check for indicators }
          while Temp_List^. Next <> Nil do
            begin
              if (Temp_List^. Next^. Indicator)
                then begin
                  Temp_Element2 := Temp_List^. Next;
                  Temp_List^. Next := Temp_List^. Next^. Next;
                  if Temp_Element2 = Upward_Embed_Graph^ [Current_Vertex]. Edges
                    then Upward_Embed_Graph^ [Current_Vertex]. Edges := Temp_Element2^. Next;
                  if not (Temp_Element2^. Same_As)        { check if we agree with direction }
                    then Upward_Embed_Graph^
                           [Temp_Element2^. Indicator_Vertex]. Reverse := true
                end
                else Temp_List := Temp_List^. Next    { and check next edge }
            end
        end
    end
end;

{-------------------------------------------------------}
{- An indicator has been found in the PQ-subtree       -}
{-  that is to be replaced during vertex addition.     -}
{- We note this fact in the adjacency list of the vertex.  -}
{-------------------------------------------------------}
procedure Note_Indicator_Presence (Current_Indicator,
                                   Left_Sibling       : PQ_Node_Ptr);
var
  Temp_Embed_Edge : Embed_Edge_Ptr;

begin
  new (Temp_Embed_Edge);              { add to adjacency list }
  Temp_Embed_Edge^. Next :=
      Upward_Embed_Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges;
  Upward_Embed_Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges :=
      Temp_Embed_Edge;
  Temp_Embed_Edge^. Indicator := true;    { note it is an indicator }
  Temp_Embed_Edge^. Indicator_Vertex := Current_Indicator^. For_Vertex;  { and check direction }
  Temp_Embed_Edge^. Same_As := (Current_Indicator^.Left_Sib^. Element = Left_Sibling);
end;
```

```
{-------------------------------------------------------}
{- Explore the subtree of T rooted at Node.            -}
{- In particular we are searching for leaves and indicators.-}
{-                                                     -}
{- Note that this DFS_Explore is never FIRST called with  -}
{- an indicator. Thus, we can ignore the indicators that  -}
{- are not the children of Q-nodes.                    -}
{-------------------------------------------------------}
procedure DFS_Explore (Node : PQ_Node_Ptr);

var
  Temp_Embed_Edge : Embed_Edge_Ptr;
  Temp_Node,
  Temp_Node2      : PQ_Node_Ptr;
  Temp_Double     : Double_Ptr;

begin
  if Node^. Node_Type = Q_Node
    then begin                                  { walk every child of Node }
      Temp_Node := Node^. Rightmost_Kid;
      Temp_Node2 := Node^. Rightmost_kid^. Immediate_Siblings^. Element;
      DFS_Explore (Temp_Node);
      while Temp_Node2 <> Nil do
        begin
          if Temp_Node2^. Node_Type = Indicator   { then since part of full subtree }
            then                                  {  the direction is unimportant }
              If Temp_Node2^. Left_Sib^. Element <> Temp_Node
                then Upward_Embed_Graph^ [Temp_Node2^. For_Vertex].   { merely note direction }
                                    Reverse := true;
          DFS_Explore (Temp_Node2);
          Walk_Normal (Temp_Node, Temp_Node2)      { and get next sibling }
        end
    end
    else
      if Node^. Node_Type = P_Node
        then begin                              { for every child do }
          Temp_Double := Node^. List_Start;
          if Temp_Double <> Nil
            then repeat
                   DFS_Explore (Temp_Double^. Element);  { explore it }
                   Temp_Double := Temp_Double^. Right
                 until Temp_Double = Node^. List_Start
        end
        else if Node^. Node_Type = Leaf         { add the leaf to the adjacency list }
               then begin
                 new (Temp_Embed_Edge);
                 Temp_Embed_Edge^. Indicator := false;
                 Temp_Embed_Edge^. Tail_Vertex := Node^. Tail_Vertex;
                 Temp_Embed_Edge^. Next :=
                   Upward_Embed_Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges;
                 Upward_Embed_Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges :=
                   Temp_Embed_Edge
               end
               else begin         { we ignore the indicator }
                 if Node^. Immediate_Siblings <> Nil
                   then Kill_Node (Node)
               end
end;
```

```
{-------------------------------------------------------------}
{- The nodes affected during this pass of the algorithm will-}
{- be reset to initial values.                               -}
{-------------------------------------------------------------}
procedure Reset_Marked_Vertices;

var
  List_Element,
  List_Element2 : List_Ptr;

begin                              { Reset Marked Vertices }
  List_Element := Used_List;
  while List_Element <> Nil do     { For every used element }
    begin
      if List_Element^. Element <> Pseudo_Node
        then
          if List_Element^. Element^. Data_Label <> Full   { If it was not Full }
            then with List_Element^. Element^ do
                   begin                                   { Reset all the relevant fields }
                     Full_Kids := Nil;
                     Partial_Kids := Nil;
                     Full_Kids_Count := 0;
                     Data_Label := Empty;
                     Mark := None;
                     Pert_Leaf_Count := 0;
                     Pert_Child_Count := 0
                   end
            else;                                          { For a full node get memory }
      List_Element2 := List_Element;
      List_Element := List_Element^. Next;                 { go to next element }
    end;
  Used_List := Nil                                         { Note no new elements }
end;
```

```
{-------------------------------------------------------}
{- Perform the actual vertex addition stage.           -}
{-------------------------------------------------------}
procedure Remove_Full_Kids_Insert_new_Node (Current_Node,
                                           New_Node,
                                           Start_Node_Sib,
                                           End_Node_Sib,
                                           Start_Next,
                                           End_Prev    : PQ_Node_Ptr);
var
  New_Indicator,            { the new indicator to insert }
  Temp_Node,
  Full_Sib, Imm_Full_Sib,
  Empty_Sib, Imm_Empty_Sib,
  Full_Kid,
  Empty_Kid,
  Left_Kid,
  Right_Kid  : PQ_Node_Ptr;
  Temp_List  : List_Ptr;

begin
  Left_Kid := Start_Next;
  Right_Kid := End_Prev;
  Get_Full_Empty_Siblings (Right_Kid, Full_Sib, Imm_Full_Sib,
                           Empty_Sib, Imm_Empty_Sib);   { Get the neighbours }

              { ---------- The next section of code deletes the sequence of full kids ------------}

  if Right_Kid = Left_Kid                              { one kid is special }
    then begin
      if (Empty_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Right_Kid, Imm_Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node);
      if (Full_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Right_Kid, Imm_Full_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node)
    end
    else begin
      if (Empty_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Right_Kid, Imm_Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Right_Kid, New_Node);
      Get_Full_Empty_Siblings (Left_Kid, Full_Sib, Imm_Full_Sib,
                               Empty_Sib, Imm_Empty_Sib);   { Get the neighbours }
      if (Empty_Sib <> Nil)
        then Add_Replace_Sibling (New_Node, Left_Kid, Imm_Empty_Sib)
        else Adjust_End_most_Kids (Current_Node, Left_Kid, New_Node)
    end;

  Create_PQ_Node (New_Indicator);              { insert a new indicator in the sequeuce of children }
  New_Indicator^. Node_Type := Indicator;
  New_Indicator^. For_Vertex := ST_Number_Index [Num_Vertices_Added];
  if Start_Node_Sib <> Nil
    then begin
      Add_Replace_Sibling (New_Indicator, New_Node, Start_Node_Sib);
      Add_Replace_Sibling (New_Indicator, Start_Node_Sib, New_Node);
      with New_Indicator^ do
        if Immediate_Siblings^. Element = Start_Node_Sib
          then Left_Sib := Immediate_Siblings
```

```
          else Left_Sib := Immediate_Siblings^. Next
    end
    else begin
      Add_Replace_Sibling (New_Indicator, New_Node, End_Node_Sib);
      Add_Replace_Sibling (New_Indicator, End_Node_Sib, New_Node);
      with New_Indicator^ do
        if Immediate_Siblings^. Element = New_Node
          then Left_Sib := Immediate_Siblings
          else Left_Sib := Immediate_Siblings^. Next
    end
end;


{-------------------------------------------------------------}
{- Special case where Root of pertinent subtree is a Pseudo -}
{- Node.                                                     -}
{-------------------------------------------------------------}
procedure Add_New_Q_Root_Pseudo_Node (var New_Root : PQ_Node_Ptr);

var
  Finished   : Boolean;
  Start_Node_Sib,
  End_Node_Sib,
  Full_Sib, Imm_Full_Sib,
  Empty_sib, Imm_Empty_Sib,
  Start_Next, End_Prev,
  Temp_Node,
  Temp_Node2 : PQ_Node_Ptr;
  Temp_List  : List_Ptr;

begin
  Temp_List := Root_Node^. Full_Kids;
  Finished := false;
  while not Finished do                          { find an endmost full kid }
    begin
      if (NBour_of (Temp_List^. Element)^. Data_Label <> Full)
       or (Other_NBour_of (Temp_List^. Element)^. Data_Label <> Full)
        then Finished := true
        else Temp_List := Temp_List^. Next
    end;
  Temp_Node := Temp_List^. Element;
                                          { and traverse the list of sibs }
  Get_Full_Empty_Siblings (Temp_Node, Full_Sib, Imm_Full_Sib,
                           Empty_Sib, Imm_Empty_sib);
  Start_Node_Sib := Imm_Empty_Sib;
  Temp_Node2 := Imm_Full_Sib;
  Start_Next := Temp_Node;
  DFS_Explore (Temp_Node);                  { performing a DFS to get edges, indicators }
  while (Temp_Node2 <> Nil)
    and ((Temp_Node2^. Node_Type = Indicator)
          or (Temp_Node2^. Data_Label = Full)) do
    begin
      if Temp_Node2^. Node_Type = Indicator
        then Note_Indicator_Presence (Temp_Node2, Temp_Node)   { add indicator }
        else DFS_Explore (Temp_Node2);                         { else check subtree }
      Walk_Normal (Temp_Node, Temp_Node2);                     { get next edge }
    end;
  End_Prev := Temp_Node;
```

```
  End_Node_Sib := Temp_Node2;

  Create_PQ_Node (New_Root);    { New node to insert }
  with New_Root^ do
    begin
      Node_Type := P_Node;
      List_Start := Nil;
      Child_Count := 0
    end;
        { since Pseudo Node we know that there are empty sibs to either side, so just insert }

  Remove_Full_Kids_Insert_new_Node (Pseudo_Node, New_Root,        { and remove full kids, insert new P-node }
                                    Start_Node_Sib, End_Node_Sib,
                                    Start_Next, End_Prev);
  Add_List_to_Used_List (Root_Node^. Full_Kids);             { Wipe the kid }
  Add_Node_to_Used_List (Root_Node);                         { and wipe the Pseudo node }
end;

{-----------------------------------------------------------}
{- An ordinary Q-root replacement.                          -}
{-----------------------------------------------------------}
procedure Add_New_Q_Root (var New_Root : PQ_Node_Ptr);

var
  List_Element : List_Ptr;
  Start_Node_Sib,
  End_Node_Sib,
  Temp_Node2,
  Temp_Node,
  Full_Sib, Imm_Full_Sib,
  Empty_Sib, Imm_Empty_Sib,
  Start_Next, End_Prev,
  New_Indicator,
  Sibling1,
  Sibling2     : PQ_Node_Ptr;
  Left,
  Finished     : Boolean;

begin
  List_Element := Root_Node^. Full_Kids;
  Finished := false;
  while not Finished do           { find endmost full kid }
    begin
      Temp_Node := NBour_of (List_Element^. Element);
      Temp_Node2 := Other_NBour_of (List_Element^. Element);
      if (Temp_Node^. Data_Label <> Full) or (Temp_Node2 = Nil)
          or (Temp_Node2^. Data_Label <> Full)
        then Finished := true
        else List_Element := List_Element^. Next
    end;
  Temp_Node := List_Element^. Element;
  Left := (List_Element^. Element = Root_Node^. Leftmost_Kid);
  DFS_Explore (Temp_Node);
  Get_Full_Empty_Siblings (Temp_Node, Full_Sib, Imm_Full_Sib,
                           Empty_Sib, Imm_Empty_sib);
  Start_Node_Sib := Imm_Empty_Sib;
  Start_Next := Temp_Node;
  Temp_Node2 := Imm_Full_Sib;
```

```
  while (Temp_Node2 <> Nil)
    and ((Temp_Node2^. Node_Type = Indicator)
          or (Temp_Node2^. Data_Label = Full)) do        { check each sib }
    begin
      if Temp_Node2^. Node_Type = Indicator
        then Note_Indicator_Presence (Temp_Node2, Temp_Node)  { add if indicator }
        else DFS_Explore (Temp_Node2);                        { else DFS to check subtree }
      Walk_Normal (Temp_Node, Temp_Node2);
    end;
  End_Prev := Temp_Node;
  End_Node_Sib := Temp_Node2;

  if (Start_Node_Sib <> nil) or (End_Node_Sib <> nil)     { now form new node to insert }
    then begin                                            { but must check if not full Q-node }
      Create_PQ_Node (New_Root);    { create New node to insert }
      with New_Root^ do
        begin
          Node_Type := P_Node;      { reset values }
          Parent := Root_Node;
          List_Start := Nil;
          Child_Count := 0
        end;
      Remove_Full_Kids_Insert_new_Node (Root_Node, New_Root,    { and remove full kids, insert new P-node }
                                        Start_Node_Sib, End_Node_Sib,
                                        Start_Next, End_Prev);

    end
    else begin
      New_Root := Root_Node;
      with New_Root^ do
        begin
          Node_Type := P_Node;
          List_Start := Nil;
          Child_Count := 0
        end;
    end;
  Root_Node^. Data_Label := Empty;
  Add_List_to_Used_List (Root_Node^. Full_Kids);  { Reset values }
  Add_Node_to_Used_List (Root_Node);
  Temp_Node := Root_Node^. Parent;                { And add all the parents }
  while Temp_Node <> Nil do                       { to be reset as well     }
    begin
      Add_Node_To_Used_List (Temp_Node);
      Temp_Node := Temp_Node^. Parent
    end
end;
```

```
{-------------------------------------------------------}
{- Straight forward replacement of the P node with new root -}
{- We do not have to worry about the indicators at this     -}
{- level. Only worry if an indicator was in a subtree and   -}
{- a child of a Q-node                                      -}
{-------------------------------------------------------}
procedure Add_New_P_Root (var New_Root : PQ_Node_Ptr);

var
  Temp_Node    : PQ_Node_Ptr;
  Temp_Double2,
  Temp_Double  : Double_Ptr;

begin
  DFS_Explore (Root_Node);
  if Root_Node^. List_Start <> Nil    { wipe all the children }
    then begin
      Temp_Double := Root_Node^. List_Start^. Right;
      while Temp_Double <> Root_Node^. List_Start do
        begin
          Temp_Double2 := Temp_Double;
          Temp_Double := Temp_Double^. Right;
        end;
      Root_Node^. List_Start := Nil
    end;
  Root_Node^. Data_Label := Empty;
  Root_Node^. Child_Count := 0;
  Add_List_to_Used_List (Root_Node^. Full_Kids);   { Reset the values }
  Add_Node_to_Used_List (Root_Node);
  Temp_Node := Root_Node^. Parent;                 { and reset the values of the parents }
  while Temp_Node <> Nil do
    begin
      Add_Node_To_Used_List (Temp_Node);
      Temp_Node := Temp_Node^. Parent
    end;
  New_Root := Root_Node
end;
```

```
{-------------------------------------------------------}
{- Main embedding algorithm.                           -}
{-------------------------------------------------------}
begin
  Global_Initialise_Lempel;              { Do main initialisation }
  if Is_2_Connected
    then begin
      Planar := true;                    { Assume the graph is planar }
      Current_Vertex := 1;
      Initialise_LEC (Current_Vertex);                    { Initialise a tree and ST-Numbering }
      while (Num_Vertices_Added <= Last_Vertex - 1 - Num_Deleted) and (Planar) do
        begin
          Num_Vertices_Added := Num_Vertices_Added + 1;
          if Debug_Embedding
            then writeln ('Bunching ST number ', Num_vertices_Added, ' now...');
          Bubble_Tree;                   { Do the Bubble - Pass I     }
          Reduce_Pertinent_Subtree;      { Do the Reduction - Pass II }
          if Planar                      { Replace sub-tree from Root (T, S) }
            then begin
              if Root_Node^. Node_Type = Q_Node
                then if Root_Node = Pseudo_Node      { Two different cases }
                       then Add_New_Q_Root_Pseudo_Node (New_Root)
                       else Add_New_Q_Root (New_Root)
                else begin
                  if Root_Node^. Node_Type = Leaf    { leaf is a special case of P-node }
                    then begin
                      DFS_Explore (Root_Node);
                      with Root_Node^ do
                        begin
                          Node_Type := P_Node;
                          List_Start := Nil;
                          Child_Count := 0
                        end
                    end;
                  Add_New_P_Root (New_Root)          { and replace P-node with new P-node and edges }
                end;
              Reset_Marked_Vertices;       { Reset the nodes affected during bubbling }
              if Num_Vertices_Added <> Last_Vertex - Num_Deleted
                then Add_Edges_to_Tree (ST_Number_Index [Num_Vertices_Added],
                                        New_Root);   { and add the new leaves }
            end;
        end;
      if not Planar
        then begin
          writeln ('The graph is non-planar');
          prompt;
        end
        else begin
          writeln ('Generating an embedding of the graph');
          Correct_Adjacency_Lists;       { make the appropriate reversals }
          Extend_Embedding;              { and extend the embedding from upward to entire }
          Write_Embedded_Graph           { and copy the embedding to the Graph }
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit contains the Bubble phase (Pass I) of the -}
{- Reduction algorithm.                                -}
{-------------------------------------------------------}
unit Embedd_Bubble;

interface

procedure Bubble_Tree;       { single procedure to perform the bubble phase }

implementation

uses
  Embedd_Globals,
  Planar_Defs;

{-------------------------------------------------------}
{- Perform the bubble.                                 -}
{-------------------------------------------------------}
procedure Bubble_Tree;

var
  Finished           : Boolean;      { Finished the bubble }
  Blocked_List       : List_Ptr;     { List of all nodes which are blocked }
  Block_Count,                       { Blocks of blocked nodes }
  Blocked_Nodes      : Integer;      { Total number of blocked nodes }
  Loop,
  Off_the_Top        : 0..1;         { if we have reached the Root of the tree }
  Current_Parent,
  Current_Node       : PQ_Node_Ptr;
  Blocked_Siblings   : Integer;      { Number of Blocked Siblings   }

{-------------------------------------------------------}
{- Variables are initialised to their default values.  -}
{-------------------------------------------------------}
procedure Initialise_Bubble;

begin
  Blocked_List := Nil;
  Pseudo_Node := Nil;
  Queue_Start := Nil;
  Queue_Head := Nil;
  Block_Count := 0;
  Blocked_Nodes := 0;
  Off_the_Top := 0;
end;
```

```
{-------------------------------------------------------}
{- A Blocked node is added to the Blocked list.        -}
{-------------------------------------------------------}
procedure Add_to_Blocked_List (Node : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Element := Node;
  List_Element^. Next := Blocked_List;
  Blocked_List := List_Element
end;

{-------------------------------------------------------}
{- An Blocked node is now unBlocked, and must be removed -}
{- from the Blocked List.                              -}
{-------------------------------------------------------}
procedure Remove_from_Blocked_List (Node : PQ_Node_Ptr);

var
  Temp_Element,
  List_Element : List_Ptr;

begin
  if Blocked_List <> Nil
    then begin
      List_Element := Blocked_List;
      if List_Element^. Element = Node  { check if the element is at the Head of the list }
        then begin
          Blocked_List := Blocked_List^. Next;
        end
        else begin
          while List_Element^. Next^. Element <> Node do    { search for the element }
            List_Element := List_Element^. Next;
          Temp_Element := List_Element^. Next;
          List_Element^. Next := Temp_Element^. Next;       { delete from the list }
        end
    end
    else Halt                  { error - trying to remove from empty list }
end;
```

```
{-------------------------------------------------------}
{- The Bubble procedure has finished with Block_Count = 1.  -}
{- This means that the pertinent children of the Root are   -}
{- blocked and are in one group. So we give them a 'new'    -}
{- parent which we will delete at the end of the Reduction. -}
{- Hence Pseudo since it is not really their parent.        -}
{-------------------------------------------------------}
procedure Create_Pseudo_Node;

var
  Number_EndMost_Kids : Integer;    { We need the two end kids }
  List_Element        : List_Ptr;

begin
  Create_PQ_Node (Pseudo_Node);     { Create the new node. }
  With Pseudo_Node^ do
    begin
      Node_Type := Q_Node;
      Number_EndMost_Kids := 0;
      while (Blocked_List <> Nil) do     { set the endmost kids }
        begin
          List_Element := Blocked_List;
          with List_Element^. Element^. Immediate_Siblings^ do   { an endmost kid is defined }
            if (Element^. Mark <> Blocked)                       { as one who does not have  }
              or (Next^. Element^. Mark <> Blocked)              { both siblings Blocked.    }
              then begin
                if Number_EndMost_Kids = 1              { set the relevant endmost pointer }
                  then RightMost_Kid := List_Element^. Element
                  else LeftMost_Kid := List_Element^. Next^. Element;
                Number_EndMost_Kids := Number_EndMost_Kids + 1
              end;
          Pert_Child_Count := Pert_Child_Count + 1;        { Count the number of pertinent kids }
          List_Element^. Element^. Parent := Pseudo_Node;  { and set the parent pointer }
          Blocked_List := Blocked_List^. Next;
        end
    end
end;
```

```
{-------------------------------------------------------}
{- We determine status of the node - Blocked or UnBlocked   -}
{-------------------------------------------------------}
procedure Block_or_Unblock_Node;

var
  UnBlocked_Siblings,                { Number of Unblocked Siblings   }
  Dud_Siblings       : Integer;      { Number of neither Blocked nor UnBlocked }
  Temp_Node          : PQ_Node_Ptr;  { Temporary variable }

begin
  Current_Node^. Mark := Blocked;   { assume it is blocked }
  UnBlocked_Siblings := 0;          { Reset counts of neighbours }
  Dud_Siblings := 0;
  Blocked_Siblings := 0;
  with Current_Node^ do
    begin
      if Immediate_Siblings <> Nil
        then begin
          Temp_Node := Nbour_of (Current_Node);
          if Temp_Node^. Mark = Blocked
            then Blocked_Siblings := Blocked_Siblings + 1
            else
              if Temp_Node^. Mark = UnBlocked
                then UnBlocked_Siblings := UnBlocked_Siblings + 1
                else Dud_Siblings := Dud_Siblings + 1;  { neither blocked nor unblocked }
          Temp_Node := Other_Nbour_of (Current_Node);
          if Temp_Node <> Nil
            then
              if Temp_Node^. Mark = Blocked
                then Blocked_Siblings := Blocked_Siblings + 1
                else
                  if Temp_Node^. Mark = UnBlocked
                    then UnBlocked_Siblings := UnBlocked_Siblings + 1
                    else Dud_Siblings := Dud_Siblings + 1;  { neither blocked nor unblocked }
        end;
      if UnBlocked_Siblings <> 0        { Check if we may unblock the Node }
        then begin
          Temp_Node := NBour_of (Current_Node);
          if Temp_Node^. Mark <> UnBlocked
            then Temp_Node := Other_NBour_of (Current_Node);
          Parent := Temp_Node^. Parent;      { Give it a proper Parent Reference }
          Mark := UnBlocked
        end
        else
          if (UnBlocked_Siblings + Blocked_Siblings + Dud_Siblings < 2)   { if Parent is P Node }
            then Mark := UnBlocked;                                       { or endmost kid of parent }
    end
end;
```

```
{-------------------------------------------------------}
{- The adjacent block of Blocked Nodes may be unBlocked.    -}
{-------------------------------------------------------}
procedure UnBlock_Blocked_Siblings;

var
  Finished       : Boolean;
  First_Sibling,
  Second_Sibling : List_Ptr;
  Previous_Node,
  Temp_Node      : PQ_NOde_Ptr;

begin
  Temp_Node := NBour_of (Current_Node);
  if Temp_Node^. Mark <> Blocked          { get the Sibling that is Blocked }
      then Temp_Node := Other_NBour_of (Current_Node);
  Previous_Node := Current_Node;
  Finished := false;
  while not Finished do       { walk through the Block, UnBlocking them }
    begin
      Temp_Node^. Mark := UnBlocked;
      Remove_from_Blocked_List (Temp_Node);
      Blocked_Nodes := Blocked_Nodes - 1;
      Temp_Node^. Parent := Current_Parent;        { Unblock the node }
      Current_Parent^. Pert_Child_Count :=
            Current_Parent^. Pert_Child_Count + 1; { increment Pertinent Child Count }
      Walk (Previous_Node, Temp_Node);
      if Temp_Node = Nil
        then Finished := true;
      if not Finished
        then Finished := not (Temp_Node^. Mark = Blocked)   { check if we reached end of block }
    end
end;
```

```
{-------------------------------------------------------}
{- Main Bubble Procedure.                               -}
{-------------------------------------------------------}
begin
  Initialise_Bubble;          { Initialise }
  Place_Leaves_On_Queue;      { Start with the Leaves }
  if Sizeof_Queue = 1
    then Remove_from_Queue (Current_Node);  { This is already valid }
  while (Sizeof_Queue + Block_Count + Off_the_Top) > 1 do
    begin
      if Sizeof_Queue = 0
        then begin
          Planar := false;
          Exit
        end;
      Remove_from_Queue (Current_Node);       { Get the current node off the Queue }
      Block_or_Unblock_Node;                  { Determine marking for Node.        }
      if Current_Node^. Mark = UnBlocked
        then begin
          Current_Parent := Current_Node^. Parent;   { get the parent }
          if Blocked_Siblings > 0
            then UnBlock_Blocked_Siblings;           { Unblock siblings if necessary }
          if Current_Parent = Nil
            then Off_the_Top := 1                    { Note we have reached top of Tree }
            else begin
              Current_Parent^. Pert_Child_Count :=
                  Current_Parent^. Pert_Child_Count + 1;
              if Current_Parent^. Mark = None        { if we have not Queued the Parent, then do so }
                then begin
                  Add_to_Queue (Current_Parent);
                  Current_Parent^. Mark := Queued
                end
            end;
          Block_Count := Block_Count - Blocked_Siblings;  { decrement the number of blocks }
        end
        else begin                                          { A new Blocked Sibling }
          Block_Count := Block_Count + 1 - Blocked_Siblings; { Increase Block_Count appropriately }
          Blocked_Nodes := Blocked_Nodes + 1;
          Add_to_Blocked_List (Current_Node)                { and add the node to Blocked list }
        end
    end;
  if (Block_Count = 1) and (Blocked_Nodes > 1)   { Pseudo Node case }
    then Create_Pseudo_Node
    else begin
      if Block_Count > 1
        then Planar := false             { more than one block of blocked siblings }
        else if Blocked_Nodes = 1
               then;
      Pseudo_Node := Nil                 { not the Pseudo case }
    end
end;

end.
```

```
{-------------------------------------------------------------}
{- The Reduction Unit of Lempel, Even and Cederbaum.         -}
{- In this unit the Reduction phase is carried out.          -}
{-------------------------------------------------------------}
unit Embedd_Reduce;

interface

procedure Reduce_Pertinent_Subtree;   { single procedure to perform the reduction }

implementation

uses
  Planar_Defs,
  Embedd_Globals;

{-------------------------------------------------------------}
{- We perform the reduction by matching Templates.           -}
{-------------------------------------------------------------}
procedure Reduce_Pertinent_Subtree;

var
  Sizeof_Set_S    : Integer;      { The size of the pertinent leaf set }
  Current_Parent,
  Current_Node    : PQ_Node_Ptr;

{-------------------------------------------------------------}
{- Returns the number of partial kids a particular node has.-}
{-------------------------------------------------------------}
function Count_Partial_Kids (Node : PQ_Node_Ptr) : Integer;

var
  List_Element : List_Ptr;
  Count        : Integer;

begin
  List_Element := Current_Node^. Partial_Kids;
  Count := 0;
  while (List_Element <> Nil) do     { for every partial node do }
    begin
      Count := Count + 1;                  { increment the count }
      List_Element := List_Element^. Next   { and go to next partial node }
    end;
  Count_Partial_Kids := Count        { return the total }
end;
```

```
{-------------------------------------------------------}
{- A single Full child at List_Posn in the full list is     -}
{- removed from Parent_Node's kids and added to Full_Node's.-}
{-------------------------------------------------------}
procedure Remove_Add (var List_Posn    : List_Ptr;
                      var Parent_Node,
                          Full_Node    : PQ_Node_Ptr);

var Chain_Posn : Double_Ptr;

begin
  Chain_Posn := List_Posn^. Element^. Circ_List_Posn;
  if Chain_Posn^. Right = Chain_Posn                { special case, the only kid }
    then Parent_Node^. List_Start := Nil
    else begin                                      { otherwise delete the node as usual }
      Chain_Posn^. Right^. Left := Chain_Posn^. Left;
      Chain_Posn^. Left^. Right := Chain_Posn^. Right;
      if Parent_Node^. List_Start = Chain_Posn
        then Parent_Node^. List_Start := Chain_Posn^. Right;
    end;
  Add_to_Circle_Link (Full_Node, Chain_Posn^. Element);
end;

{-------------------------------------------------------}
{- The full nodes are stripped from From_Node and added as  -}
{- kids of Full_Node returned.                              -}
{-------------------------------------------------------}
procedure Strip_Full_Nodes (var From_Node, Full_Node : PQ_Node_Ptr);

var
  Temp_Double  : Double_Ptr;
  Temp_List    : List_Ptr;

begin
  if From_Node^. Full_Kids_Count > 1       { only one kid is a special case }
    then begin
      Create_PQ_Node (Full_Node);
      with Full_Node^ do
        begin
          Data_Label := Full;                         { set up values for Full Node }
          Node_Type := P_Node;
          Child_Count := From_Node^. Full_Kids_Count;
          List_Start := Nil;                          { no kids yet! }
          Temp_List := From_Node^. Full_Kids;
          while Temp_List <> Nil do                   { strip from parent and add to Full Node }
            begin
              Remove_Add (Temp_List, From_Node, Full_Node);
              Temp_List := Temp_List^. Next
            end;
          Full_Kids := From_Node^. Full_Kids;
          Full_Kids_Count := From_Node^. Full_Kids_Count;   { and copy corresponding data across }
        end
    end
    else begin                                        { only one kid }
      Full_Node := From_Node^. Full_Kids^. Element;   { so merely remove it from parent kids list }
      Temp_Double := Full_Node^. Circ_List_Posn;
      if Temp_Double^. Right = Temp_Double
        then From_Node^. List_Start := nil
```

```
        else begin
          Temp_Double^. Right^. Left := Temp_Double^. Left;
          Temp_Double^. Left^. Right := Temp_Double^. Right;
          if From_Node^. List_Start = Temp_Double
            then From_Node^. List_Start := Temp_Double^. Right
        end;
    end;
  From_Node^. Child_Count := From_Node^. Child_Count -   { adjust child count accordingly }
                                    From_Node^. Full_Kids_Count;
  From_Node^. Full_Kids_Count := 0;      { Note no full kids }
  From_Node^. Full_Kids := Nil;          { and list is empty }
end;

{-------------------------------------------------------}
{- Node1 and Node2 are made siblings.                   -}
{-------------------------------------------------------}
procedure Add_Sibling (Node1, Node2 : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Next := Node1^. Immediate_Siblings;
  List_Element^. Element := Node2;
  Node1^. Immediate_Siblings := List_Element;
  new (List_Element);
  List_Element^. Next := Node2^. Immediate_Siblings;
  List_Element^. Element := Node1;
  Node2^. Immediate_Siblings := List_Element
end;

{-------------------------------------------------------}
{- New1 had Old1 as a Sibling, now has New2.           -}
{- New2 had Old2 as a Sibling, now has New1.           -}
{-------------------------------------------------------}
procedure Replace_Replace_Siblings (New1, Old1, New2, Old2 : PQ_Node_Ptr);

begin
  with New1^. Immediate_Siblings^ do
    if Element = Old1
      then Element := New2
      else Next^. Element := New2;
  with New2^. Immediate_Siblings^ do
    if Element = Old2
      then Element := New1
      else Next^. Element := New1
end;
```

```
{-------------------------------------------------------}
{- Counts number of times a full kid that has no full  -}
{- sibling or no sibling at all (i.e. endmost).        -}
{-------------------------------------------------------}
procedure Count_End_Full_Kids (var Parent_Node : PQ_Node_Ptr; var Count : integer);

var
  Temp_Kid     : PQ_Node_Ptr;
  List_Element : List_Ptr;

begin
  Count := 0;
  List_Element := Parent_Node^. Full_Kids;
  while (Count < 3) and (List_Element <> Nil) do      { greater then 3 is equivalent to 3 }
    begin
      Temp_Kid := List_Element^. Element;
      if NBour_of (Temp_Kid)^. Data_Label <> full     { check nbours }
            then Count := Count + 1;
      Temp_Kid := Other_NBour_of (Temp_Kid);
      if (Temp_Kid = Nil) or (Temp_Kid^. Data_Label <> Full)   { and inc count if necessary }
            then Count := Count + 1;
      List_Element := List_Element^. Next
    end;
end;


{-------------------------------------------------------}
{- The Full and Empty endmost Kids of a Q node are found.   -}
{-------------------------------------------------------}
procedure Get_Full_Empty_Children (var Node,
                                   Full_Child, Empty_Child : PQ_Node_Ptr);

begin
  if Node <> Nil
    then begin
      if Node^. RightMost_Kid^. Data_Label <> Empty
        then begin
          Full_Child := Node^. RightMost_Kid;
          Empty_Child := Node^. LeftMost_Kid;
        end
        else begin
          Full_Child := Node^. LeftMost_Ki   d;
          Empty_Child := Node^. RightMost_Kid;
        end
    end
    else begin
      Full_Child := Nil;
      Empty_Child := Nil
    end
end;
```

```
{-------------------------------------------------------------}
{- To follow we code all the Templates.                      -}
{-------------------------------------------------------------}

{-------------------------------------------------------------}
{- Template L1 - A single Leaf.                              -}
{-------------------------------------------------------------}
function Template_L1 (Current_Node : PQ_Node_Ptr) : Boolean;

var List_Element : List_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> Leaf
      then Template_L1 := false
      else begin
        Data_Label := full;              { Note that the Leaf is Full }
        if Root_Node = Nil               { If we are not at the Pertinent Root }
          then begin
            Parent^. Full_Kids_Count := Parent^. Full_Kids_Count + 1; { Note an extra kid }
            new (List_Element);
            List_Element^. Element := Current_Node;       { and add to full kids list }
            List_Element^. Next := Parent^. Full_Kids;
            Parent^. Full_Kids := List_Element
          end;
        Template_L1 := true
      end;
end;

{-------------------------------------------------------------}
{- Template P1 - A P Node whose kids are all Full.           -}
{-------------------------------------------------------------}
function Template_P1 (Current_Node : PQ_Node_Ptr) : Boolean;

var List_Element : List_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> P_Node
      then Template_P1 := false
      else if Full_Kids_Count <> Child_Count   { Check if kids are all full }
             then Template_P1 := false
             else begin
               Data_Label := full;       { Note it is also full }
               if Root_Node = Nil        { Check if we are at the Pertinent Root }
                 then begin
                   Parent^. Full_Kids_Count :=        { If not we add extra full kid to parent }
                     Parent^. Full_Kids_Count + 1;
                   new (List_Element);
                   List_Element^. Element := Current_Node;
                   List_Element^. Next := Parent^. Full_Kids;
                   Parent^. Full_Kids := List_Element;
                   Add_List_to_Used_List (Current_Node^. Full_Kids);    { Add Full Kids List to Used list }
                   Current_Node^. Full_Kids := Nil;
                 end;
               Template_P1 := true
             end
end;
```

```
{-------------------------------------------------------}
{- Template P2 - Must be at the Pertinent Root - A P-Node   -}
{- some of the Kids are full, the rest are empty.           -}
{-------------------------------------------------------}
function Template_P2 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Full_Node  : PQ_Node_Ptr;

begin
  With Current_Node^ do
    if (Node_Type <> P_Node) or (Partial_Kids <> Nil)
      then Template_P2 := false
      else begin
        if Full_Kids_Count = 1            { Return that node }
          then Root_Node := Full_Kids^. Element
          else begin
            Strip_Full_Nodes (Current_Node, Full_Node);       { Delete all the full nodes from Current_Node }
            Add_to_Circle_Link (Current_Node, Full_Node);     { Re-Add Full node to Current_Node }
            Current_Node^. Child_Count := Current_Node^. Child_Count + 1;
            Full_Node^. Parent := Current_Node;
            Root_Node := Full_Node;                           { and return the new Pertinent Root }
          end;
        Add_Node_to_Used_List (Current_Node);
        Template_P2 := true
      end
end;

{-------------------------------------------------------}
{- Template P3 - Must NOT be at the Pertinent Root          -}
{- A P-Node with some Kids are full, the rest are empty.    -}
{-------------------------------------------------------}
function Template_P3 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element : List_Ptr;
  Empty_Node,
  Full_Node,
  New_Root     : PQ_Node_Ptr;
  Number_Empty : Integer;

begin
  with Current_Node^ do
    if (Node_Type <> P_Node) or (Partial_Kids <> Nil)
      then Template_P3 := false
      else begin
        Create_PQ_Node (New_Root);            { The new Parent is a Q-Node }
        New_Root^. Node_Type := Q_Node;
        Number_Empty := Child_Count - Full_Kids_Count;
        Replace_Node_Partial (Current_Node, New_Root, false);   { Replace Current_Node by New_Root }
                                                                { in Parent and Sibling chains     }
        Strip_Full_Nodes (Current_Node, Full_Node);     { Now strip all full kids from Current_Node }
        new (List_Element);
        List_Element^. Element := Full_Node;
        List_Element^. Next := New_Root^. Full_Kids;    { Add Full_Node to Full_Kids of New_Root }
        New_Root^. Full_Kids := List_Element;
        New_Root^. Full_Kids_Count := 1;                { and note count of 1 }
```

```
        Full_Node^. Parent := New_Root;                { Add Full_Node to kids }
        if Number_Empty = 1                            { Note which Node is the empty node }
          then Empty_Node := List_Start^. Element      { if there is only 1 empty, then we want }
                                                       { to avoid chains - so delete Current_Node }
          else begin
            Empty_Node := Current_Node;
            Empty_Node^. Child_Count := Number_Empty;
            Empty_Node^. Data_Label := Empty
          end;
        Empty_Node^. Parent := New_Root;               { Empty_Node's parent is New_Root }
        Add_Sibling (Empty_Node, Full_Node);           { They are Siblings }
        Full_Node^. Data_Label := Full;
        New_Root^. RightMost_Kid := Full_Node;         { They are also endmost kids }
        New_Root^. LeftMost_Kid := Empty_Node;
        if Number_Empty < 2                            { we want to avoid chains }
          then Current_Node := Nil
          else Add_Node_to_Used_List (Current_Node);
        Add_List_to_Used_List (Full_Node^. Full_Kids);
        Full_Node^. Full_Kids := Nil;
        Template_P3 := true
      end;
end;

{-----------------------------------------------------------}
{- Template P4 - Must be at the Pertinent Root, A P-Node    -}
{- with one Partial Q-Node and possibly some Kids are full, -}
{- possibly some are empty.                                 -}
{-----------------------------------------------------------}
function Template_P4 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element : List_Ptr;
  Full_Node,
  Only_Partial : PQ_Node_Ptr;

begin
  With Current_Node^ do
    if Node_Type <> P_Node
      then Template_P4 := false
      else
        if Count_Partial_Kids (Current_Node) <> 1    { Must have exactly 1 partial kid }
          then Template_P4 := false
          else begin
            Only_Partial := Partial_Kids^. Element;  { This is our new Root }
            if (Only_Partial^. LeftMost_Kid^. Data_Label <> Full)
                  and (Only_Partial^. RightMost_Kid^. Data_Label <> Full)
              then Template_P4 := false              { Must have a full element at endmost }
              else begin
                if Full_Kids_Count > 0               { Get rid of full kids on current }
                  then begin
                    Strip_Full_Nodes (Current_Node, Full_Node);
                    Add_List_to_Used_List (Full_Node^. Full_Kids);
                    Full_Node^. Full_Kids := Nil;
                    Full_Node^. Parent := Only_Partial;
                    new (List_Element);
                    List_Element^. Next := Only_Partial^. Full_Kids;   { concat lists of full kids }
                    List_Element^. Element := Full_Node;
                    Only_Partial^. Full_Kids := List_Element;
```

```
                Only_Partial^. Full_Kids_Count := Only_Partial^. Full_Kids_Count + 1;
                if Only_Partial^. Leftmost_Kid^. Data_Label = Full  { and adjust endmost accordingly }
                  then begin
                    Add_Sibling (Only_Partial^. Leftmost_Kid, Full_Node);
                    Only_Partial^. Leftmost_Kid := Full_Node
                  end
                  else begin
                    Add_Sibling (Only_Partial^. Rightmost_Kid, Full_Node);
                    Only_Partial^. Rightmost_Kid := Full_Node
                  end
              end;
            Root_Node := Only_Partial;    { Note the new pertinent Root }
            Template_P4 := true
          end
      end
end;

{-------------------------------------------------------------}
{- Template P5 - Must NOT be at the Pertinent Root, A P-Node-}
{- with one Partial Q-Node and possibly some Kids are full, -}
{- possibly some are empty.                                  -}
{-------------------------------------------------------------}
function Template_P5 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  List_Element        : List_Ptr;
  Temp_Double         : Double_Ptr;
  Number_Empty        : Integer;
  New_Root,
  Full_Node,
  Empty_Node,
  Empty_EndMost_Child,
  Full_EndMost_Child  : PQ_Node_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> P_Node
      then Template_P5 := false
      else
        if Count_Partial_Kids (Current_Node) <> 1 { must have exactly 1 Partial Kid }
          then Template_P5 := false
          else begin
            New_Root := Partial_Kids^. Element;    { this node becomes the Root of the replacement }
            Number_Empty := Current_Node^. Child_Count -
                            Current_Node^. Full_Kids_Count -
                              Count_Partial_Kids (Current_Node);
            Get_Full_Empty_Children (New_Root,     { get the endmost kids }
                                     Full_Endmost_Child, Empty_Endmost_Child);
            Replace_Node_Partial (Current_Node, New_Root, true);   { replace Current_Node with New_Root }
            if Full_Kids_Count > 0                              { get rid of full kids of Current_Node }
              then begin
                Strip_Full_Nodes (Current_Node, Full_Node);
                Add_List_to_Used_List (Full_Node^. Full_Kids);
                Full_Node^. Full_Kids := Nil;
                Full_Node^. Parent := New_Root;
                new (List_Element);
                List_Element^. Next := New_Root^. Full_Kids;       { adjust full kids accordingly }
                List_Element^. Element := Full_Node;
```

```
              New_Root^. Full_Kids := List_Element;
              New_Root^. Full_Kids_Count := New_Root^. Full_Kids_Count + 1;
              if New_Root^. Leftmost_Kid^. Data_Label = Full     { and adjust endmost kid }
                then begin
                  Add_Sibling (New_Root^. Leftmost_Kid, Full_Node);
                  New_Root^. Leftmost_Kid := Full_Node
                end
                else begin
                  Add_Sibling (New_Root^. Rightmost_Kid, Full_Node);
                  New_Root^. Rightmost_Kid := Full_Node
                end;
            end;
          if Number_Empty > 0          { Note the empty node }
            then begin
              if Number_Empty = 1      { As usual, watch for chains }
                then begin
                  Temp_Double := Current_Node^. List_Start;
                  while Temp_Double^. Element^. Data_Label <> Empty do  { get first empty node }
                    Temp_Double := Temp_Double^. Right;
                  Empty_Node := Temp_Double^. Element
                end
                else begin                    { more than 1 empty - so current_node will do }
                  Empty_Node := Current_Node;
                  Empty_Node^. Data_Label := Empty;
                  Empty_Node^. Child_Count := Number_Empty
                end;
              Empty_Node^. Parent := New_Root;                { New_Root is its new parent }
              Add_Sibling (Empty_EndMost_Child, Empty_Node);  { Add Sibling pointers }
              if New_Root^. LeftMost_Kid^. Data_Label = Empty { Adjust endmost kids }
                then New_Root^. LeftMost_Kid := Empty_Node
                else New_Root^. RightMost_Kid := Empty_Node;
            end;
          if Number_Empty < 2          { cleanup for chains }
            then Current_Node := Nil
            else Add_Node_to_Used_List (Current_Node);
          Template_P5 := true
        end
end;
```

```
{-------------------------------------------------------------}
{- Template P6 - Must be at the Pertinent Root, A P-Node     -}
{- with two Partial Q-Nodes and possibly some Kids are full,-}
{- possibly some are empty.                                  -}
{-------------------------------------------------------------}
function Template_P6 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Number_Empty      : Integer;
  Temp_List         : List_Ptr;
  Temp_Double       : Double_Ptr;
  Partial1,
  Partial2,
  Full_Node,
  Full_Kid_Partial1,
  Empty_Kid_Partial2,
  Full_Kid_Partial2 : PQ_Node_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> P_Node
      then Template_P6 := false
      else
        if Count_Partial_Kids (Current_Node) <> 2
          then Template_P6 := false
          else begin
            Partial1 := Partial_Kids^. Element;          { Partial1 becomes the New_Root }
            Partial2 := Partial_Kids^. Next^. Element;
            Number_Empty := Child_Count - Full_Kids_Count - 2;
            if Full_Kids_Count > 0                        { strip Full_Nodes off Current_Node }
              then begin
                Strip_Full_Nodes (Current_Node, Full_Node);
                Add_List_to_Used_List (Full_Node^. Full_Kids);
                new (Temp_List);
                Temp_List^. Element := Full_Node;
                Temp_List^. Next := Partial1^. Full_Kids;
                Partial1^. Full_Kids := Temp_List;
                Partial1^. Full_Kids_Count := Partial1^. Full_Kids_Count + 1;
              end
              else Full_Node := Nil;
            Get_Full_Empty_Children (Partial2,           { get Partial2's endmost kids }
                                     Full_Kid_Partial2, Empty_Kid_Partial2);
            Empty_Kid_Partial2^. Parent := Partial1;      { will become Partial1's endmost empty }
            if Partial1^. LeftMost_Kid^. Data_Label = Full     { adjust endmost kid pointers }
              then begin
                Full_Kid_Partial1 := Partial1^. LeftMost_Kid;
                Partial1^. LeftMost_Kid := Empty_Kid_Partial2
              end
              else begin
                Full_Kid_Partial1 := Partial1^. RightMost_Kid;
                Partial1^. RightMost_Kid := Empty_Kid_Partial2
              end;
            if Full_Node = Nil
              then Add_Sibling (Full_Kid_Partial1, Full_Kid_Partial2)
              else begin
                Add_Sibling (Full_Kid_Partial1, Full_Node);
                Add_Sibling (Full_Kid_Partial2, Full_Node)
              end;
```

```
            Temp_Double := Partial2^. Circ_List_Posn;          { delete partial2 from circ link }
            Temp_Double^. Right^. Left := Temp_Double^. Left;
            Temp_Double^. Left^. Right := Temp_Double^. Right;
            Child_Count := Child_Count - 1;                    { Count declines accordingly }
            if List_Start = Temp_Double
              then List_Start := Temp_Double^. Right;
            Temp_List := Partial2^. Full_Kids;
            while Temp_List^. Next <> Nil do
              Temp_List := Temp_List^. Next;
            Temp_List^. Next := Partial1^. Full_Kids;
            Partial1^. Full_Kids := Partial2^. Full_Kids;
            Partial1^. Full_Kids_Count := Partial1^. Full_Kids_Count + Partial2^. Full_Kids_Count;
            Root_Node := Partial1;
            if Number_Empty = 0                                { Check for chains }
              then begin
                Replace_Node_Partial (Current_Node, Root_Node, false);   { Replace with Root_Node }
                Current_Node := Nil
              end
              else Add_Node_to_Used_List (Current_Node);
            Template_P6 := true
          end;
end;

{-------------------------------------------------------------}
{- Template Q1 - A Q-Node whose kids are all full.          -}
{-------------------------------------------------------------}
function Template_Q1 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Temp_List : List_Ptr;
  Count     : integer;

begin
  with Current_Node^ do
    if (Node_Type <> Q_Node) or (Current_Node = Pseudo_Node)    { Pseudo Node is always Q3 }
      then Template_Q1 := false
      else begin
        if (RightMost_Kid^. Data_Label <> Full)
          or (LeftMost_Kid^. Data_Label <> Full)
          then Template_Q1 := false
          else begin
            Count_end_Full_Kids (Current_Node, Count);
            if (Count <> 2) or (Rightmost_Kid^. Data_Label <> Full)
                  or (Leftmost_Kid^. Data_Label <> Full)
              then Template_Q1 := false
              else begin
                Data_Label := full;                    { all kids are full, so Current_Node is full }
                if Root_Node = Nil
                  then begin                           { update Parent's full kids data }
                    Parent^. Full_Kids_Count :=
                      Parent^. Full_Kids_Count + 1;
                    new (Temp_List);
                    Temp_List^. Element := Current_Node;    { add current to full kids }
                    Temp_List^. Next := Parent^. Full_Kids; { of the parent           }
                    Parent^. Full_Kids := Temp_List;
                  end;
                Template_Q1 := true
              end
```

```
          end
      end
end;

{-------------------------------------------------------}
{- Template Q2 - A Q-Node some of whose kids are full,      -}
{-                        at most one partial kid,         -}
{-                        and some kids are empty.         -}
{- If there are any Full kids, they must all be at one end. -}
{- The partial kid MUST follow the full kids.              -}
{-------------------------------------------------------}
function Template_Q2 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Count         : Integer;
  End_Kid,
  Partial_Child,
  Full_Child,
  Empty_Child,
  Imm_Full_sib,
  Full_Sibling,
  Imm_Empty_Sib,
  Empty_Sibling : PQ_Node_Ptr;
  Temp_List     : List_Ptr;

begin
  with Current_Node^ do
    if (Node_Type <> Q_Node) or (Current_Node = Pseudo_Node)    { Pseudo Node is Q3 }
      then Template_Q2 := false
      else
        if Count_Partial_Kids (Current_Node) > 1     { at most 1 partial kid }
          then Template_Q2 := false
          else begin
            if Partial_Kids <> Nil
              then Partial_Child := Partial_Kids^. Element   { get the full kid }
              else Partial_Child := Nil;
            if Full_Kids_Count > 0
              then begin
                Count_end_Full_Kids (Current_Node, Count);
                if RightMost_Kid^. Data_Label = Full       { get the endmost kid }
                  then End_Kid := RightMost_Kid
                  else
                    if LeftMost_Kid^. Data_Label = Full
                      then End_Kid := LeftMost_Kid
                      else End_Kid := Nil;
                if (End_Kid = Nil)
                         or (Count <> 2)      { illegal no endmost kids or more than 2 }
                  then begin
                    Template_Q2 := false;
                    exit
                  end;
                if Partial_Child <> Nil
                  then begin
                    Get_Full_Empty_Siblings (Partial_Child,
                                             Full_Sibling, Imm_Full_Sib,
                                             Empty_Sibling, Imm_Empty_Sib);
                    if (Full_Sibling^. Data_Label <> Full) { if there is a partial kid, }
                      then begin                           { then it must be neighbour  }
```

```
                Template_Q2 := false;
                Exit
              end;
          end
      end
      else      { there are no Full Kids }
        if (Partial_Child <> RightMost_Kid)    { Check if the Partial kid is endmost }
          and (Partial_Child <> LeftMost_Kid)
          then begin
            Template_Q2 := false;
            Exit
          end
          else Get_Full_Empty_Siblings (Partial_Child,
                                        Full_Sibling, Imm_Full_sib,
                                        Empty_Sibling, Imm_Empty_sib);

    Data_Label := Partial;    { Everything is now checked, Template Q2 is matched }
    new (Temp_List);
    Temp_List^. Next := Parent^. Partial_Kids;   { Add to Partial list of parent }
    Temp_List^. Element := Current_Node;
    Parent^. Partial_Kids := Temp_List;
    if Partial_Child <> Nil    { must join sibling links between Partial Child's }
      then begin               { kids and Partial_Child's Siblings               }
        Get_Full_Empty_Children (Partial_Child,
                                 Full_Child, Empty_Child);   { Get endmost kids }
        if Full_Sibling <> Nil
          then
            Add_Replace_Sibling (Full_Child, Partial_Child,
                                 Imm_Full_Sib)
          else begin
            if RightMost_Kid = Partial_Child     { adjust endmost pointers   }
              then RightMost_Kid := Full_Child   { to be Partial_Child's kid }
              else LeftMost_Kid := Full_Child;
            Full_Child^. Parent := Current_Node  { set proper parent pointer }
          end;
        if Empty_Sibling <> Nil                  { set up link to Partial Child's kid }
          then Add_Replace_Sibling (Empty_Child, Partial_Child,
                                    Imm_Empty_Sib)
          else begin                             { if no empty Sibling then }
            if RightMost_Kid = Partial_Child     { there are only full kids }
              then RightMost_Kid := Empty_Child  { so Partial_Child was endmost }
              else LeftMost_Kid := Empty_Child;
            Empty_Child^. Parent := Current_Node { set new endmost Parent pointer }
          end;
        if Full_Kids <> Nil
          then begin
            Temp_List := Partial_Child^. Full_Kids;
            while Temp_List^. Next <> Nil do
              Temp_List := Temp_List^. Next;
            Temp_List^. Next := Full_Kids;
            Full_Kids := Partial_Child^. Full_Kids;
            Full_Kids_Count := Full_Kids_Count + Partial_Child^. Full_Kids_Count
          end
          else begin
            Full_Kids := Partial_Child^. Full_Kids;
            Full_Kids_Count := Partial_Child^. Full_Kids_Count
          end;
        Kill_Node (Partial_Child)
```

```
            end;
          Template_Q2 := true
        end
end;

{-------------------------------------------------------}
{- Template Q3 - A Q-Node some of whose kids are full,      -}
{-                       at most two partial kids,         -}
{-                       and some kids are empty.          -}
{- If there are any Full kids, they must all be between the -}
{- two partial kids, or next to the one.                   -}
{-------------------------------------------------------}
function Template_Q3 (Current_Node : PQ_Node_Ptr) : Boolean;

var
  Count             : Integer;
  Partial_Child_1,
  Partial_Child_2,
  Imm_Empty_Sib1, Empty_Sibling1,
  Imm_Full_Sib1, Full_Sibling1,
  Imm_Empty_Sib2, Empty_Sibling2,
  Imm_Full_Sib2, Full_Sibling2,
  Full_Child,
  Empty_Child       : PQ_Node_Ptr;
  Temp_List         : List_Ptr;

begin
  with Current_Node^ do
    if Node_Type <> Q_Node
      then Template_Q3 := false
      else
        if Count_Partial_Kids (Current_Node) > 2   { at most 2 partial kids }
          then Template_Q3 := false
          else begin
            if Partial_Kids <> Nil        { set up the partial kids }
              then begin
                Partial_Child_1 := Partial_Kids^. Element;
                if Partial_Kids^. Next <> Nil
                  then Partial_Child_2 := Partial_Kids^. Next^. Element
                  else Partial_Child_2 := Nil
              end
              else Partial_Child_1 := Nil;
            if (Full_Kids_Count > 0)
              then begin
                Count_end_Full_Kids (Current_Node, Count);
                if (Count > 2)      { check for separated full kids }
                  then begin
                    Template_Q3 := false;
                    Exit
                  end;
                if Partial_Child_1 <> Nil
                  then begin
                    Get_Full_Empty_Siblings (Partial_Child_1,
                                             Full_Sibling1, Imm_Full_Sib1,
                                             Empty_Sibling1, Imm_Empty_Sib1);
                    if Full_Sibling1^. Data_Label <> Full
                      then begin
                        Template_Q3 := false;
```

```
                exit
              end;
            if Partial_Child_2 <> Nil
              then begin
                Get_Full_Empty_Siblings (Partial_Child_2,
                                         Full_Sibling2, Imm_Full_Sib2,
                                         Empty_Sibling2, Imm_Empty_Sib2);
                if Full_Sibling2^. Data_Label <> Full
                  then begin
                    Template_Q3 := false;
                    exit
                  end;
              end
          end
      end
      else   { No full kids }
        if Count_Partial_Kids (Current_Node) = 2   { then they must be adjacent }
          then begin
            Get_Full_Empty_Siblings (Partial_Child_1,
                                     Full_Sibling1, Imm_Full_Sib1,
                                     Empty_Sibling1, Imm_Empty_sib1);
            if Full_Sibling1 <> Partial_Child_2
              then begin
                Template_Q3 := false;
                Exit
              end
          end;
    if Partial_Kids <> Nil       { nothing to do if no partial kids }
      then begin
        Data_Label := Partial;   { Template match is Confirmed }
        Get_Full_Empty_Children (Partial_Child_1,
                                 Full_Child, Empty_Child);
        if Empty_Sibling1 <> Nil      { make the links }
          then Add_Replace_Sibling (Empty_Child, Partial_Child_1,
                                    Imm_Empty_Sib1)
          else Adjust_End_Most_Kids (Current_Node, Partial_Child_1,
                                     Empty_Child);
        if Full_Sibling1 <> Nil      { make the links }
          then Add_Replace_Sibling (Full_Child, Partial_Child_1,
                                    Imm_Full_Sib1)
          else Adjust_End_Most_Kids (Current_Node, Partial_Child_1, Full_Child);
        if Partial_Child_2 <> Nil      { do the same for Partial2 }
          then begin
            Get_Full_Empty_Siblings (Partial_Child_2,
                                     Full_Sibling2, Imm_Full_Sib2, { may have changed }
                                     Empty_Sibling2, Imm_Empty_sib2);
            Get_Full_Empty_Children (Partial_Child_2,
                                     Full_Child, Empty_Child);
            if Empty_Sibling2 <> Nil
              then Add_Replace_Sibling (Empty_Child, Partial_Child_2,
                                        Imm_Empty_Sib2)
              else Adjust_End_Most_Kids (Current_Node, Partial_Child_2, Empty_Child);
            if Full_Sibling2 <> Nil      { make the links }
              then Add_Replace_Sibling (Full_Child, Partial_Child_2,
                                        Imm_Full_Sib2)
              else Adjust_End_Most_Kids (Current_Node, Partial_Child_2, Full_Child);
          end;
        if Full_Kids = Nil
```

```
                then begin
                  Full_Kids := Partial_Child_1^. Full_Kids;
                  Full_Kids_Count := Partial_Child_1^. Full_Kids_Count
                end
                else begin
                  Temp_List := Partial_Child_1^. Full_Kids;
                  while Temp_List^. Next <> nil do
                    Temp_List := Temp_List^. Next;
                  Temp_List^. Next := Full_Kids;
                  Full_Kids := Partial_Child_1^. Full_Kids;
                  Full_Kids_Count := Full_Kids_Count + Partial_Child_1^. Full_Kids_Count
                end;
              if Partial_Child_2 <> Nil
                then begin
                  Temp_List := Partial_Child_2^. Full_Kids;
                  while Temp_List^. Next <> nil do
                    Temp_List := Temp_List^. Next;
                  Temp_List^. Next := Full_Kids;
                  Full_Kids := Partial_Child_2^. Full_Kids;
                  Full_Kids_Count := Full_Kids_Count + Partial_Child_2^. Full_Kids_Count
                end;
              Kill_Node (Partial_Child_1);    { cleanup memory allocations }
              if Partial_Child_2 <> Nil
                then Kill_Node (Partial_Child_2)
            end;
          Template_Q3 := true
        end;
end;

{-------------------------------------------------------------}
{- Main Initialise Routine                                    -}
{-------------------------------------------------------------}
procedure Initialise_Reduction;

var
  Dummy_PQ     : PQ_Node_Ptr;

begin
  if Queue_Start <> Nil
    then repeat
      Remove_from_Queue (Dummy_PQ)     { Cleanup after Bubble Phase }
    until Queue_Start = Nil;
  Root_Node := Nil
end;
```

```
begin
  Initialise_Reduction;                          { Do main Init }
  Place_Leaves_On_Queue;                         { Place pertinent leaves on Queue }
  Sizeof_Set_S := Sizeof_Queue;
  while (Sizeof_Queue > 0) and (Planar) do
    begin
      Remove_from_Queue (Current_Node);
      if Current_Node^. Node_Type = Leaf
        then Current_Node^. Pert_Leaf_Count := 1;
      if Current_Node^. Pert_Leaf_Count < Sizeof_Set_S    { Check if we are at pertinent Root }
        then begin                        { if not, then }
          Current_Parent := Current_Node^. Parent;
          Current_Parent^. Pert_Leaf_Count :=     { update Parent's Pert_Leaf_Count of Pertinent Leaves }
            Current_Parent^. Pert_Leaf_Count + Current_Node^. Pert_Leaf_Count;
          Current_Parent^. Pert_Child_Count :=    { Update number of kids left to process }
            Current_Parent^. Pert_Child_Count - 1;
          if Current_Parent^. Pert_Child_Count = 0  { see if all kids matched }
            then Add_to_Queue (Current_Parent);
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P3 (Current_Node) then
          if not Template_P5 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
            begin
              writeln ('Halting no template matches');
              Planar := false
            end
        end
        else begin       { This node is the Pertinent Root }
          Root_Node := Current_Node;              { default is this node is Pertinent Root }
          if not Template_L1 (Current_Node) then
          if not Template_P1 (Current_Node) then
          if not Template_P2 (Current_Node) then
          if not Template_P4 (Current_Node) then
          if not Template_P6 (Current_Node) then
          if not Template_Q1 (Current_Node) then
          if not Template_Q2 (Current_Node) then
          if not Template_Q3 (Current_Node) then
            begin
              writeln ('Halting no template matches at pertinent root');
              Planar := false
            end
        end
    end
end;

end.
```

```
{-------------------------------------------------------}
{- This unit contains the routines use by both the Bubble   -}
{- and Reduction units of Lempel, Even and Cederbaum.       -}
{-------------------------------------------------------}
unit Embedd_Globals;

Interface

uses
  CRT,
  Planar_Miscellaneous,
  Planar_Defs;

{-------------------------------------------------------}
{- All the global variables.                            -}
{-------------------------------------------------------}
var
  ST_Number_Index    : Array [1..Max_Vertices] of Vertex_Ptr;  { used to map a St Number to }
                                                               { an actual vertex number    }
  Used_List          : List_Ptr;          { used to reset values after each reduction }
  Pseudo_Node,                            { see chapter 4 for details }
  New_Root,                               { the new root node to add new leaves to after a reduction }
  Root_Node          : PQ_Node_Ptr;       { the Root of the pertinent tree after a reduction }
  Sizeof_Queue       : Integer;           { Queue size for bubble and reduction }
  Current_Vertex,                         { Used for planarity testing of the graph }
  Num_Vertices_Added : Vertex_Ptr;        {                    ""                   }
  Queue_Start,                            { For the Queue                }
  Queue_Head         : Double_Ptr;

procedure Write_Embedded_Graph;                          { dumps embedded graph for debug purposes }

procedure Kill_Node (var Node : PQ_Node_Ptr);            { resets values on a now redundant node }

procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr); { usual sense of chain traversal }

procedure Walk (var Previous, Current : PQ_Node_Ptr);    { walk to next sibling given last two }

function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;    { find a nbour of }

function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;  { find the other nbour of }

procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);  { creates a new node with defaults }

procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);          { adds a node to the queue }

procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr);  { removes a node from the queue }

procedure Place_Leaves_on_Queue;                         { adds all pertinent leaves to the queue }

procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;  { adds a child to a P node parent }
                              var Child_Ptr : PQ_Node_Ptr);

procedure Add_Edges_to_Tree (Where_From  : Vertex_Ptr;    { adds new leaves to the tree }
                             Tree_Node   : PQ_Node_Ptr);

procedure Add_Replace_Sibling (New_Node, Old_Node, Other_Node : PQ_Node_Ptr);
                         { adds Other_Node to New_Node's Sibling list and }
                         { replaces Old_Node with New_Node in Other_Node's Sibling list }
```

```
procedure Get_Full_Empty_Siblings (var Node,
                                   Full_Sibling, Imm_Full_Sib,
                                   Empty_Sibling, Imm_Empty_Sib : PQ_Node_Ptr);
                        { Gets the full/partial sibling and empty sibling of a node }

procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);   { adds a single node to a used list }

procedure Add_List_to_Used_List (var List_Start : List_Ptr);      { adds a list of nodes to used list }

procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);
                        { Replaces in the Parent all references to Old_Node by New_Node }


Implementation

{-------------------------------------------------------------}
{- Returns a PQ_Node initialised with default values        -}
{- Will not initialise fields dependent on node type.       -}
{-------------------------------------------------------------}
procedure Create_PQ_Node (var PQ_Pointer : PQ_Node_Ptr);

begin
 new (PQ_Pointer);
 with PQ_Pointer^ do                { set the initial values }
   begin
     Data_Label := Empty;
     Full_Kids := Nil;
     Partial_Kids := Nil;
     Immediate_Siblings := Nil;
     Mark := None;
     Parent := Nil;
     Child_Count := 0;
     Full_Kids_Count := 0;
     Pert_Child_Count := 0;
     Pert_Leaf_Count := 0;
   end
end;
```

```
{-------------------------------------------------------}
{- Adds a PQ node to the Queue                          -}
{-------------------------------------------------------}
procedure Add_to_Queue (PQ_Element : PQ_Node_Ptr);

var Queue_Element : Double_Ptr;

begin
  new (Queue_Element);
  Queue_Element^. Left := Nil;
  Queue_Element^. Right := Queue_Start;
  Queue_Element^. Element := PQ_Element;
  if Queue_Start <> Nil
    then Queue_Start^. Left := Queue_Element;
  Queue_Start := Queue_Element;
  if Queue_Head = Nil
    then Queue_Head := Queue_Element;
  Sizeof_Queue := Sizeof_Queue + 1
end;

{-------------------------------------------------------}
{- Removes a PQ node from the Queue                     -}
{-------------------------------------------------------}
procedure Remove_from_Queue (var PQ_Element : PQ_Node_Ptr);

var  Queue_Element : Double_Ptr;

begin
  if Queue_Start = Nil
    then Halt;
  Queue_Element := Queue_Head;
  Queue_Head := Queue_Head^. Left;
  if Queue_Start^. Right = Nil
    then Queue_Start := Nil;
  if Queue_Head <> Nil
    then Queue_Head^. Right := Nil;
  PQ_Element := Queue_Element^. Element;
  Sizeof_Queue := Sizeof_Queue - 1
end;

{-------------------------------------------------------}
{- Adds all pertinent leaves to the Queue.              -}
{-------------------------------------------------------}
procedure Place_Leaves_on_Queue;

var Temp_Edge : Edge_Ptr;

begin
  Temp_Edge := Graph^ [ST_Number_Index [Num_Vertices_Added]]. Edges;  { get start edge }
  while (Temp_Edge <> Nil) do
    begin
      if (Graph^ [Temp_Edge^. Vertex]. ST_Number < Num_Vertices_Added)  { if it is part of the }
        then Add_to_Queue (Temp_Edge^. PQ_Ptr);                         { graph so far then add to queue }
      Temp_Edge := Temp_Edge^. Next
    end
end;
```

```
{-------------------------------------------------------}
{- Adds Child_Ptr to the circular list of Parent_Node.      -}
{-------------------------------------------------------}
procedure Add_to_Circle_Link (var Parent_Node : PQ_Node_Ptr;
                              var Child_Ptr : PQ_Node_Ptr);
var Temp_Double : Double_Ptr;

begin
  with Parent_Node^ do
    if List_Start = Nil           { check for initial case }
      then begin
        new (List_Start);
        with List_Start^ do       { set up single element circular list }
          begin
            Left := List_Start;
            Right := List_Start;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := List_Start
          end;
      end
      else begin
        new (Temp_Double);
        with Temp_Double^ do      { normal insertion into circular list }
          begin
            Left := List_Start^. Left;
            Right := List_Start;
            List_Start^. Left^. Right := Temp_Double;
            List_Start^. Left := Temp_Double;
            Element := Child_Ptr;
            Child_Ptr^. Circ_List_Posn := Temp_Double
          end
      end
end;
```

```
{-------------------------------------------------------}
{- Replace all references of Old_Node with New_Node in      -}
{- Parent. Replace it in Sibling Chains and Circular list if-}
{- necessary. Delete_Node indicates if New_Node is a child  -}
{- of Old_Node and whether it must be deleted from Old_Node -}
{- circular list.                                           -}
{- Also add New_Node to the partial children list of parent.-}
{-------------------------------------------------------}
procedure Replace_Node_Partial (Old_Node, New_Node : PQ_Node_Ptr;
                                Delete_Node : Boolean);

var                              { all variables are temporary }
  Temp_Element2,
  List_Element   : List_Ptr;
  Double_Element : Double_Ptr;
  Temp_Node      : PQ_Node_Ptr;

begin
  New_Node^. Parent := Old_Node^. Parent;     { Get new parent }
  New_Node^. Pert_Leaf_Count := Old_Node^. Pert_Leaf_Count;
  New_Node^. Data_Label := Partial;        { new node is partial }
  if Root_Node = Nil                       { so add to Partial children list }
    then begin
      new (List_Element);
      List_Element^. Next := New_Node^. Parent^. Partial_Kids;
      List_Element^. Element := New_Node;
      New_Node^. Parent^. Partial_Kids := List_Element
    end;
  if Delete_Node      { We must delete Old_Node from circular list of Old_Node }
    then begin
      Double_Element := New_Node^. Circ_List_Posn;
      if Double_Element^. Left = Double_Element    { check for end condition }
        then Old_Node^. List_Start := Nil
        else begin                                 . { otherwise delete as normal }
          if Double_Element = Old_Node^. List_Start
            then Old_Node^. List_Start := Double_Element^. Right;
          Double_Element^. Left^. Right := Double_Element^. Right;
          Double_Element^. Right^. Left := Double_Element^. Left;
        end;
      New_Node^. Circ_List_Posn := Nil;
      Old_Node^. Child_Count := Old_Node^. Child_Count - 1;  { note a child has been removed }
    end;
  if Old_Node^. Immediate_Siblings = Nil    { then Parent is a P Node and replace }
    then begin                              { in parent's circulra list.          }
      New_Node^. Immediate_Siblings := Nil;
      New_Node^. Circ_List_Posn := Old_Node^. Circ_List_Posn;
      New_Node^. Circ_List_Posn^. Element := New_Node;
    end
    else begin                              { Replace in immediate sibling's Siblings chain }
      New_Node^. Immediate_Siblings :=
           Old_Node^. Immediate_Siblings;
      Old_Node^. Immediate_Siblings := Nil;
      List_Element := New_Node^. Immediate_Siblings;   { with each sibling }
      while (List_Element <> Nil) do
        begin
          Temp_Node := List_Element^. Element;
          Temp_Element2 := Temp_Node^. Immediate_Siblings;    { search fro refernce to Old_Node }
          while (Temp_Element2^. Element <> Old_Node) do
```

```
          Temp_Element2 := Temp_Element2^. Next;
        Temp_Element2^. Element := New_Node;              { replace with New_Node }
        List_Element := List_Element^. Next          { and go to next sibling }
      end;
    if New_Node^. Parent^. LeftMost_Kid = Old_Node          { adjust end most pointers as well }
      then New_Node^. Parent^. LeftMost_Kid := New_Node
      else if New_Node^. Parent^. RightMost_Kid = Old_Node
            then New_Node^. Parent^. RightMost_Kid := New_Node;
  end;
end;

{-------------------------------------------------------}
{- Add new leaves representing edges from From_Vertex.      -}
{-------------------------------------------------------}
procedure Add_Edges_to_Tree (Where_From : Vertex_Ptr;
                             Tree_Node  : PQ_Node_Ptr);

var
  New_Nodes       : Integer;           { used to check for only a single leaf }
                                       { added since we don't want chains     }
  Last_Edge,
  From_Temp_Edge  : Edge_Ptr;
  Current_ST      : Vertex_Ptr_Range;
  PQ_Element      : PQ_Node_Ptr;

begin
  New_Nodes := 0;
  From_Temp_Edge := Graph^ [Where_From]. Edges;
  Current_ST := Graph^ [Where_From]. ST_Number;
  while From_Temp_Edge <> Nil do                  { with each edge do }
    begin
      if Graph^ [From_Temp_Edge^. Vertex]. ST_Number > Current_ST   { check it goes to a higher vertex }
        then begin
          Create_PQ_Node (PQ_Element);          { create the new leaf }
          New_Nodes := New_Nodes + 1;
          with PQ_Element^ do                   { mark it as a leaf }
            begin
              Node_Type := Leaf;
              Parent := Tree_Node;
              Tail_Vertex := Where_From
            end;
          Tree_Node^. Child_Count := Tree_Node^. Child_Count + 1;   { add to Tree Node's kids }
          Add_to_Circle_Link (Tree_Node, PQ_Element);
          From_Temp_Edge^. PQ_Ptr := PQ_Element;              { and note which PQ Element reps this edge }
          Last_Edge := From_Temp_Edge;
          From_Temp_Edge^. Other_Edge^. PQ_Ptr := PQ_Element   { ditto for other edge element }
        end;
      From_Temp_Edge := From_Temp_Edge^. Next    { go to the next candidate edge }
    end;
  if New_Nodes = 0       { Error - ST Numbering guarantees we always can add an edge }
    then begin
      writeln ('Fatal error - no new edges to add');
      halt
    end
    else
      if New_Nodes = 1              { check for chains of a single element added }
        then with Tree_Node^ do                 { tree node becomes the leaf }
          begin
```

```
            PQ_Element := List_Start^. Element;
            Node_Type := Leaf;                    { note this vertex is a leaf }
            Last_Edge^. PQ_Ptr := Tree_Node;       { change references to this node }
            Last_Edge^. Other_Edge^. PQ_Ptr := Tree_Node;
            Tail_Vertex := PQ_Element^. Tail_Vertex;
          end
end;

{-------------------------------------------------------}
{- Adds Other_Node to New_Node's Sibling List.          -}
{- Also Replaces reference in Other_Node's Sibling list to  -}
{- Old_Node with New_Node.                              -}
{-------------------------------------------------------}
procedure Add_Replace_Sibling (New_Node, Old_Node,
                               Other_Node : PQ_Node_Ptr);
var
  List_Element : List_Ptr;

begin
  List_Element := Other_Node^. Immediate_Siblings;    { Replace phase }
  if List_Element^. Element = Old_Node
    then List_Element^. Element := New_Node
    else List_Element^. Next^. Element := New_Node;
  new (List_Element);
  List_Element^. Next := New_Node^. Immediate_Siblings;   { add reference to Other_Node }
  List_Element^. Element := Other_Node;
  New_Node^. Immediate_Siblings := List_Element
end;

{-------------------------------------------------------}
{- The Full and Empty Siblings of a node are returned.      -}
{- For the purposes of this routine, a full node is either  -}
{- strictly full, or partial.                           -}
{-------------------------------------------------------}
procedure Get_Full_Empty_Siblings (var Node,
                                   Full_Sibling, Imm_Full_Sib,
                                   Empty_Sibling, Imm_Empty_Sib : PQ_Node_Ptr);

var
  Imm_Temp,
  Imm_Temp2,
  Temp,
  Temp_Node,
  Temp_Node2 : PQ_Node_Ptr;

begin
  if Node <> Nil                    { if nil then we do nothing }
    then begin
      Imm_Temp := Node^.Immediate_Siblings^.Element;    { get proper sibling }
      if Imm_Temp^. Node_Type = Indicator
        then begin
          Temp_Node := Imm_Temp;
          Temp_Node2 := Node;
          repeat
            Walk (Temp_Node2, Temp_Node);
          until Temp_Node^. Node_Type <> Indicator       { i.e. we dont want an indicator }
        end
        else Temp_Node := Imm_Temp;
```

```
      Temp := Temp_Node;
      if Node^. Immediate_Siblings^. Next = Nil          { now get other sibling }
        then Imm_Temp2 := nil
        else Imm_Temp2 := Node^. Immediate_Siblings^. Next^. Element;
      if Imm_Temp2 <> Nil
        then
          if Imm_Temp2^. Node_Type = Indicator
            then begin
              Temp_Node := Imm_Temp2;
              Temp_Node2 := Node;
              repeat
                Walk (Temp_Node2, Temp_Node);
              until Temp_Node^. Node_Type <> Indicator    { that is not an indicator }
            end
            else Temp_Node := Imm_Temp2;
      if Temp^. Data_Label <> Empty
        then begin
          Full_Sibling := Temp;
          Imm_Full_Sib := Imm_Temp;
          if Imm_Temp2 <> Nil
            then begin
              Imm_Empty_Sib := Imm_Temp2;
              Empty_Sibling := Temp_Node
            end
            else begin
              Imm_Empty_Sib := Nil;
              Empty_Sibling := Nil
            end
        end
        else begin
          Empty_Sibling := Temp;
          Imm_Empty_Sib := Imm_Temp;
          if Imm_Temp2 <> Nil
            then begin
              Imm_Full_Sib := Imm_Temp2;
              Full_Sibling := Temp_Node
            end
            else begin
              Imm_Full_Sib := Nil;
              Full_Sibling := Nil
            end
        end
    end
    else begin                                  { if the node is nil then return empties }
      Imm_Empty_Sib := Nil;
      Imm_Full_Sib := Nil;
      Empty_Sibling := Nil;
      Full_Sibling := Nil
    end
end;
```

```
{-------------------------------------------------------}
{- If one of the Endmost kids was Old_Node, then we reset   -}
{- that kid to be New_Kid.                                  -}
{-------------------------------------------------------}
procedure Adjust_End_Most_Kids (Current_Node, Old_Kid, New_Kid : PQ_Node_Ptr);

begin
  with Current_Node^ do
    if RightMost_Kid = Old_Kid
      then begin
        RightMost_Kid := New_Kid;
        New_Kid^. Parent := Current_Node;
      end
      else
        if LeftMost_Kid = Old_Kid
          then begin
            LeftMost_Kid := New_Kid;
            New_Kid^. Parent := Current_Node
          end
end;

{-------------------------------------------------------}
{-  A List is added to the Used List.                   -}
{-------------------------------------------------------}
procedure Add_List_to_Used_List (var List_Start : List_Ptr);

var
  List_Element : List_Ptr;

begin
  if List_Start <> Nil
    then begin
      List_Element := List_Start;
      while List_Element^. Next <> Nil do     { go to the end of the list }
        List_Element := List_Element^. Next;
      List_Element^. Next := Used_List;       { Append the list to the front of Used list }
      Used_List := List_Start;
      List_Start := Nil
    end
end;

{-------------------------------------------------------}
{- A single node is appended to used list.              -}
{-------------------------------------------------------}
procedure Add_Node_to_Used_List (PQ_Element : PQ_Node_Ptr);

var
  List_Element : List_Ptr;

begin
  new (List_Element);
  List_Element^. Next := Used_List;
  List_Element^. Element := PQ_Element;
  Used_List := List_Element
end;
```

```
{-------------------------------------------------------}
{- Get next sibling regardless if it is an indicator or not -}
{-------------------------------------------------------}
procedure Walk_Normal (var Previous_Node, Current_Node : PQ_Node_Ptr);

var
  Temp_Node : PQ_Node_Ptr;

begin
  Temp_Node := Current_Node;
  with Current_Node^. Immediate_siblings^ do
    if Element = Previous_Node
      then if Next = Nil
             then Current_Node := Nil
             else Current_Node := Next^. Element
      else Current_Node := Element;
  Previous_Node := Temp_Node
end;

{-------------------------------------------------------}
{- Get next sibling that is not an indicator           -}
{-------------------------------------------------------}
procedure Walk (var Previous, Current : PQ_Node_Ptr);

var
  Temp_Previous,
  Temp_Node1,
  Temp_Node2  : PQ_Node_Ptr;

begin
  with Current^. Immediate_Siblings^ do       { get the two immediate siblings }
    begin
      Temp_Node1 := Element;
      if Next <> Nil
        then Temp_Node2 := Next^. Element
        else Temp_Node2 := nil
    end;
  if Temp_Node2 = Nil
    then Current := Nil
    else begin
      Temp_Previous := Current;
      while (Temp_Node1 <> Nil) and (Temp_Node1 <> Previous)    { get sibling not an indicator }
              and (Temp_Node1^. Node_Type = Indicator) do
        Walk_Normal (Temp_Previous, Temp_Node1);
      if (Temp_Node1 <> Nil) and (Temp_Node1 = Previous)        { we can now decide on the direction }
        then begin
          Temp_Previous := Current;
          while (Temp_Node2 <> Nil) and (Temp_Node2^. Node_Type = Indicator) do
            Walk_Normal (Temp_Previous, Temp_Node2);            { and get next sibling now }
          Previous := Current;
          Current := Temp_Node2
        end
        else begin
          Previous := Current;                        { else we can just get the next sibling }
          Current := Temp_Node1
        end
    end
end;
```

```
{-------------------------------------------------------}
{- Get the first neighbour of a node that is not an indicator}
{-------------------------------------------------------}
function Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

var
  Temp_Node,
  Old_Node  : PQ_Node_Ptr;

begin
  Temp_Node := Node^. Immediate_Siblings^. Element;
  if Temp_Node^. Node_Type = Indicator
    then begin
      Old_Node := Node;
      Walk (Old_Node, Temp_Node)
    end;
  NBour_of := Temp_Node;
end;

{-------------------------------------------------------}
{- Get the second neighbour of a node that is not an       -}
{- indicator.                                              -}
{-------------------------------------------------------}
function Other_Nbour_of (Node : PQ_Node_Ptr) : PQ_Node_Ptr;

var
  Temp_Node,
  Old_Node   : PQ_Node_Ptr;

begin
  if Node^. Immediate_Siblings^. Next = Nil
    then Other_NBour_of := Nil
    else begin
      Temp_Node := Node^. Immediate_Siblings^. Next^. Element;
      if Temp_Node^. Node_Type = Indicator
        then begin
          Old_Node := Node;
          Walk (Old_Node, Temp_Node)
        end;
      Other_NBour_of := Temp_Node;
    end
end;

{-------------------------------------------------------}
{- A Q node is disposed of.                               -}
{- Since we are using Mark and Release, we do nothing except-}
{- to reset the nodes value.                              -}
{-------------------------------------------------------}
procedure Kill_Node (var Node : PQ_Node_Ptr);

begin
  Node := Nil                          { and reset its value }
end;
```

```
{-------------------------------------------------------}
{- Wipe the current graph, by reseting the edges field.  -}
{-------------------------------------------------------}
procedure Scratch_Embedded_Graph;

var
  Loop       : Embed_Vertex_Ptr;

begin
  if Last_Vertex > 0
    then
      For Loop := 1 to Last_Vertex do
        Embed_Graph^ [Loop]. Edges := Nil
end;

{-------------------------------------------------------}
{- The current graph is displayed on the screen         -}
{- Then, the embedding is written to the Graph (as different-}
{- from the Embed_Graph).                               -}
{-------------------------------------------------------}
procedure Write_Embedded_Graph;

var
  Filename          : string;
  Temp_Embed_Vertex : Embed_Vertex_Ptr;
  Temp_Embed_Edge   : Embed_Edge_Ptr;
  Temp_Edge         : Edge_Ptr;
  Temp_Vertex       : Vertex_Ptr;
  Scr               : integer;


{-------------------------------------------------------}
{- Sort the adjacency lists of Graph according to the rank  -}
{- in the adjacency list of the embedding.              -}
{- Thus, if an edge has rank 2 then the edge in the graph   -}
{- appear second in the adjacency list.                 -}
{-                                                      -}
{- We use the same sort as for egde weightings in the DFS.  -}
{-------------------------------------------------------}
procedure Sort_Adjacency_Lists;

type
  Bucket_Array_Ptr  = ^Bucket_Array_Type;
  Bucket_Array_Type = array [Vertex_Ptr] of Bucket_Ptr;

var
  Bucket_Array : Bucket_Array_Ptr;
  Temp_Bucket,
  Temp_Bucket2 : Bucket_Ptr;
  Temp_Vertex  : Integer;
  Temp_Edge    : Edge_Ptr;

begin
  new (Bucket_Array);
  For Temp_Vertex := 1 to Last_Vertex do
    Bucket_Array^ [Temp_Vertex] := Nil;
  For Temp_Vertex := 1 to Last_Vertex do
    begin
```

```
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while (Temp_Edge <> Nil) do
        begin
          Temp_Edge^. Weight := Temp_Edge^. Rank;
          new (Temp_Bucket);
          Temp_Bucket^. Data := Temp_Edge;
          Temp_Bucket^. Next := Bucket_Array^ [Temp_Edge^. Weight];
          Temp_Bucket^. Vertex := Temp_Vertex;
          Bucket_Array^ [Temp_Edge^. Weight] := Temp_Bucket;
          Temp_Edge := Temp_Edge^. Next
        end;
      Graph^ [Temp_Vertex]. Edges := Nil;
    end;
  For Temp_Vertex := Last_Vertex downto 1 do
    begin
      Temp_Bucket := Bucket_Array^ [Temp_Vertex];
      while Temp_Bucket <> Nil do
        begin
          Temp_Bucket^. Data^. Next := Graph^ [Temp_Bucket^. Vertex]. Edges;
          Graph^ [Temp_Bucket^. Vertex]. Edges := Temp_Bucket^. Data;
          Temp_Bucket2 := Temp_Bucket;
          Temp_Bucket := Temp_Bucket^. Next;
        end
    end
end;

{-------------------------------------------------------------}
{- We assign ranks to the edges.                              -}
{-------------------------------------------------------------}
procedure Rank_Adjacency_Lists;

var
  Rank_Array : Array [1..Max_Vertices] of byte;
  Rank       : Byte;
  Loop       : Vertex_Ptr;
  Temp_Edge  : Edge_Ptr;
  Temp_Embed_Edge  : Embed_Edge_Ptr;

begin
  For Loop := 1 to Last_Vertex do       { for each vertex in Graph }
    if not Graph^ [Loop]. Deleted
      then begin
        Rank := 1;
        Temp_Embed_Edge := Embed_Graph^ [Loop]. Edges;
        while Temp_Embed_Edge <> nil do { set up rank array for the embed edges }
          begin
            Rank_Array [Temp_Embed_Edge^. Tail_Vertex] := Rank;
            Rank := Rank + 1;
            Temp_Embed_Edge := Temp_Embed_Edge^. Next
          end;
        Temp_Edge := Graph^ [Loop]. Edges;  { and then assign ranks to the graph edges }
        while Temp_Edge <> nil do
          begin
            Temp_Edge^. Rank := Rank_Array [Temp_Edge^. Vertex];
            Temp_Edge := Temp_Edge^. Next
          end
      end
end;
```

```
{-------------------------------------------------------------}
{- Main Write_Embedding procedure                             -}
{-------------------------------------------------------------}
begin
  if Debug_Embedding
    then begin                  { display the embedding }
      clrscr;
      Scr := 0;
      writeln ('The Embedding of the Graph is ');
      writeln;
      writeln ('Vertex Number    Edges');
      For Temp_Vertex := 1 to Last_Vertex do  { for every vertex display the edges }
        begin
          write (Temp_Vertex:7);
          Temp_Embed_Edge := Embed_Graph^ [Temp_Vertex]. Edges;
          write (' ':8);
          while Temp_Embed_Edge <> Nil do
            begin
              write (Temp_Embed_Edge^. Tail_Vertex:4);
              Temp_Embed_Edge := Temp_Embed_Edge^. Next
            end;
          writeln;
          Scr := Scr + 1;
          if Scr = 20     { check for full screen and wait if necessary }
            then begin
              Prompt;
              Scr := 0;
            end;
        end;
      writeln ('and now back to fun! ....');
      writeln;
      prompt
    end;
  Rank_Adjacency_Lists;       { find the rank of the edges in the embeddding }
  Sort_Adjacency_Lists;       { sort the edges of the graph according to the ranking }
  Scratch_Embedded_Graph      { and wipe the embedding }
end;


end.
```

```
{-------------------------------------------------------}
{- This unit represents the Hopcroft & Tarjan algorithm.    -}
{- For a complete discussion of the method and Data         -}
{- structures used, please see Appendix A.                  -}
{-------------------------------------------------------}
unit Hop_Algorithm;

interface

uses
  Planar_Defs,
  Planar_Miscellaneous,
  Component_Handling,
  Printer,
  CRT;


procedure Hopcroft_Tarjan_Split_Components;          { the Hopcroft and Tarjan split component procedure }


implementation


{-------------------------------------------------------}
{- Returns the DFS number of the vertex.                    -}
{-------------------------------------------------------}
function Num (Vert : Vertex_Ptr_Range) : Vertex_Ptr_Range;

begin
  if Vert > 0
    then Num := Graph^ [Vert]. Number
    else Num := 0
end;

{-------------------------------------------------------}
{- The Depth-First Search (DFS) is as per Ch. 1.  We need   -}
{- perform a DFS in order to generate the lowpoints L1 and  -}
{- L2 which are essential for the algorithm to work.        -}
{- Also the Number of Descendants for a vertex are found.   -}
{-------------------------------------------------------}
procedure DFS;

{-------------------------------------------------------}
{- Returns the minimum of two numbers                       -}
{-------------------------------------------------------}
function Min (x, y : Integer) : Integer;

begin
  if Num (x) < Num(y)
    then Min := x
    else Min := y;
end;
```

```
var
  Current_Label : 0..MaxInt;  { Current DFS number assigned }
  Temp_Edge   : Edge_Ptr;     { temporary variable     }
  Finished,                   { with the algorithm     }
  New_Vertex  : Boolean;      { indicates if a new     }
                              { label must be assigned }
  Temp_Vertex2,
  Temp_Vertex : Vertex_Ptr;   { temporary variable     }

begin
  For Temp_Vertex := 1 to Last_Vertex do     { Initialise the vertex }
    with Graph^ [Temp_Vertex] do
      begin
        Used := false;                       { have not reached the vertex yet }
        Number := 0;                         { no DFS number                   }
        Father := 0;                         { no father                       }
        L1 := 0;                             { no weightings                   }
        L2 := 0;
        Num_Descend := 0;
        Temp_Edge := Edges;
        While Temp_Edge <> Nil do            { for every incident edge         }
          begin
            Temp_Edge^. Used := False;       { we have not traversed the edge  }
            Temp_Edge^. Deleted := False;    { and it has not been directed    }
            Temp_Edge := Temp_Edge^. Next
          end
      end;
  Finished := False;                         { Initial settings }
  New_Vertex := true;
  Temp_Vertex := 1;
  while Graph^ [Temp_Vertex]. Deleted do
    Temp_Vertex := Temp_Vertex + 1;
  Current_Label := 0;
  repeat                  { until all vertices have been exhausted }
    repeat                  { until all edges of current vertex explored }
      {2}
      if New_Vertex   { if we need to assign initial labellings }
        then begin
          Current_Label := Current_Label + 1;
          Graph^ [Temp_Vertex]. Number := Current_Label;  { next label }
          Graph^ [Temp_Vertex]. L1 := Temp_Vertex;      { default weightings }
          Graph^ [Temp_Vertex]. L2 := Temp_Vertex;
          Graph^ [Temp_Vertex]. Num_Descend := 1          { descendant of itself }
        end;
      {3}
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      While (Temp_Edge <> Nil) and ((Temp_Edge^. Used) or (Temp_Edge^. Deleted)) do
        Temp_Edge := Temp_Edge^. Next;
      if (Temp_Edge <> Nil)               { we have an unexplored edge }
        then begin
          {4}
            Temp_Edge^. Used := True;                 { note it is explored }
            Temp_Edge^. Other_Edge^. Deleted := true; { and direct the edge }
            with Graph^ [Temp_Edge^. Vertex] do
            if Number <> 0                            { The vertex has been visited before }
              then begin                              { so it is a back edge-adjust L1, L2 }
                if Number < Num(Graph^ [Temp_Vertex]. L1)
                  then begin
```

```
                Graph^ [Temp_Vertex]. L2 := Graph^ [Temp_Vertex]. L1;
                Graph^ [Temp_Vertex]. L1 := Temp_Edge^. Vertex;
              end
              else
                if Number > Num(Graph^ [Temp_Vertex]. L1)
                  then Graph^ [Temp_Vertex]. L2 :=
                         Min (Temp_Edge^. Vertex, Graph^ [Temp_Vertex]. L2)
                  else;
            New_Vertex := false;
          end
          else begin                              { the edge is a tree edge }
            Father := Temp_Vertex;                { add to the tree         }
            Temp_Vertex := Temp_Edge^. Vertex;    { and move to the vertex  }
            New_Vertex := true;
          end;
      end
  until (Temp_Edge = Nil);      { if Temp_Edge = Nil then all edges explored }
                                { and we need to backtrack to the father     }
  if Graph^ [Temp_Vertex]. Number = 1   { we are at the root - backtrack not possible }
    then
      {5} Finished := true
    else begin
      {6}
      with Graph^ [Graph^ [Temp_Vertex]. Father] do    { adjust father's weightings              }
        if Num(Graph^ [Temp_Vertex]. L1) < Num(L1)     { based on kid's (more recent) weightings }
          then begin
            L2 := Min (Graph^ [Temp_Vertex]. L2, L1);
            L1 := Graph^ [Temp_Vertex]. L1
          end
        else
          if Graph^ [Temp_Vertex]. L1 = L1
            then L2 := Min (Graph^ [Temp_Vertex]. L2, L2)
            else L2 := Min (Graph^ [Temp_Vertex]. L1, L2);
      Temp_Vertex2 := Temp_Vertex;
      Temp_Vertex := Graph^ [Temp_Vertex]. Father;     { and backtrack }
      Graph^ [Temp_Vertex]. Num_Descend :=
          Graph^ [Temp_Vertex]. Num_Descend +
              Graph^ [Temp_Vertex2]. Num_Descend;
      New_Vertex := false;
    end
  until Finished;
end;
```

```
{-------------------------------------------------------}
{- We reorder the edge lists of each vertex so that edges   -}
{- with a lower L1 weighting come first                     -}
{-                                                          -}
{- To do the actual sort, we place each edge in the graph   -}
{- into a bucket that represents its weighting. Then, we    -}
{- reconstruct the edge lists of each vertex by adding the  -}
{- edges from the highest bucket to the edges lists first.  -}
{- We add the edges to the front of the edge lists. Then we -}
{- proceed in order, considering lower buckets.             -}
{-------------------------------------------------------}
procedure ReOrder_Lists;

{-------------------------------------------------------}
{- Returns a weighting for the edge. Please see chapter 3   -}
{- for a justification of the weighting.                    -}
{-------------------------------------------------------}
function Phi (u, v : Vertex_Ptr) : Integer;

begin
  if Graph^ [v]. Number < Graph^ [u]. Number
    then Phi := 2 * Graph^ [v]. Number
    else if Num(Graph^ [v]. L2) >= Graph^ [u]. Number
           then Phi := 2 * Num(Graph^ [v]. L1)
           else Phi := 2 * Num(Graph^ [v]. L1) + 1
end;

var
  Bucket_Array : array [1..2 * Max_Vertices + 1] of Bucket_Ptr;
  Temp_Bucket,
  Temp_Bucket2 : Bucket_Ptr;
  Temp_Vertex  : Integer;
  Temp_Edge    : Edge_Ptr;

begin
  For Temp_Vertex := 1 to 2 * Last_Vertex + 1 do  { no edges in any bucket }
    Bucket_Array [Temp_Vertex] := Nil;
  For Temp_Vertex := 1 to Last_Vertex do          { for every vertex }
    begin
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      while (Temp_Edge <> Nil) do                 { for each edge of that vertex }
        begin
          Temp_Edge^. Virtual_Number := 0;
          Temp_Edge^. Weight := Phi (Temp_Vertex, Temp_Edge^. Vertex); { get the weigth         }
          new (Temp_Bucket);                                           { get new bucket element }
          Temp_Bucket^. Data := Temp_Edge;
          Temp_Bucket^. Next := Bucket_Array [Temp_Edge^. Weight];     { and add element to the }
          Temp_Bucket^. Vertex := Temp_Vertex;                         { correct bucket         }
          Bucket_Array [Temp_Edge^. Weight] := Temp_Bucket;
          Temp_Edge := Temp_Edge^. Next                                { go to next edge        }
        end;
      Graph^ [Temp_Vertex]. Edges := Nil;                              { reset edge list        }
    end;
  For Temp_Vertex := 2 * Last_Vertex + 1 downto 1 do      { having sorted the edges   }
    begin                                                 { we add them back in order }
      Temp_Bucket := Bucket_Array [Temp_Vertex];
      while Temp_Bucket <> Nil do                 { while edges are in the bucket }
        begin
```

```
          Temp_Bucket^. Data^. Next := Graph^ [Temp_Bucket^. Vertex]. Edges; { add the edge }
          Graph^ [Temp_Bucket^. Vertex]. Edges := Temp_Bucket^. Data;
          Temp_Bucket2 := Temp_Bucket;                                    { remove the element }
          Temp_Bucket := Temp_Bucket^. Next;                              { from the bucket    }
        end
    end
end;

{-------------------------------------------------------------}
{- The second DFS procedure, as described in Section 3.2     -}
{- The vertices are renumbered in the order that they are    -}
{- visited so that the vertex visited last has num p         -}
{- Also Degree, Highpt and A1 of each vertex are found.      -}
{-------------------------------------------------------------}
procedure Second_DFS (var Start_Vertex : Vertex_Ptr);

var
  Temp_Edge   : Edge_Ptr;
  Mapping     : array [1..Max_Vertices]     { gives the new number for the vertex }
                  of Vertex_Ptr_Range;
  Temp_Vertex,
  Last_Used   : integer;                    { last number assigned }

{-------------------------------------------------------------}
{- Recursive procedure does a DFS and generates the new      -}
{- numbering scheme.                                         -}
{-------------------------------------------------------------}
procedure Path_Finding (Current_Vertex : Vertex_Ptr);

var
  Temp_Edge   : Edge_Ptr;
  Next_Vertex : Vertex_Ptr;
  New_Number  : integer;

begin
  New_Number := Last_Used - Num_Deleted - Graph^ [Current_Vertex]. Num_Descend + 1;   { allocate new number}
  Mapping [Current_Vertex] := New_Number;                                  { Store the number }

  Temp_Edge := Graph^ [Current_Vertex]. Edges;
  while Temp_Edge <> nil do                                { for each edge incident with Current_Vertex }
    begin
      if not Temp_Edge^. Deleted                { if it is a valid directed edge }
        then begin
          Graph^ [Current_Vertex].Degree :=          { degree increases for both }
             Graph^ [Current_Vertex]. Degree + 1;    { current vertex and the    }
          Graph^ [Temp_Edge^. Vertex]. Degree :=     { destination vertex.       }
             Graph^ [Temp_Edge^. Vertex]. Degree + 1;
          Next_Vertex := Temp_Edge^. Vertex;
          if Graph^ [Current_Vertex]. Number < Graph^ [Next_Vertex]. Number  { if tree edge }
            then begin
              Path_Finding (Next_Vertex);           { then recurse the DFS }
              Last_Used := Last_Used - 1            { and exit noting allocation occured }
            end
            else
              if Graph^ [Next_Vertex]. High_Pt = 0                { otherwise update High_Pt }
                then Graph^ [Next_Vertex]. High_Pt := New_Number;
        end;
      Temp_Edge := Temp_Edge^. Next                { do next edge }
```

```
    end
end;

begin     { second DFS }
  Last_Used := Last_Vertex;                    { first number to use }
  For Temp_Vertex := 1 to Last_Vertex do       { assign defaults }
    begin
      Graph^ [Temp_Vertex]. High_Pt := 0;
      Graph^ [Temp_Vertex]. Degree := 0;
      Mapping [Temp_Vertex] := 0
    end;
  Temp_Vertex := 1;
  While Graph^ [Temp_Vertex]. Deleted do
    Temp_Vertex := Temp_Vertex + 1;
  Path_Finding (Temp_Vertex);                  { call the DFS at top of tree }
  for Temp_Vertex := 1 to Last_Vertex do       { now adjust new degree, A1 }
    if not Graph^ [Temp_Vertex]. Deleted
      then begin
        Temp_Edge := Graph^ [Temp_Vertex]. Edges;
        while (Temp_Edge^. Deleted) do
          Temp_Edge := Temp_Edge^. Next;
        Graph^ [Temp_Vertex]. A1 := Num(Temp_Edge^. Vertex);      { get first edge in adj list }
        Graph^ [Temp_Vertex]. Number := Mapping [Temp_Vertex];    { set new number }
        if Graph^ [Temp_Vertex]. Number = 1                       { and store the start vertex }
          then Start_Vertex := Temp_Vertex;
      end
end;

{-------------------------------------------------------------}
{- The main algorithm                                        -}
{-------------------------------------------------------------}
procedure Hopcroft_Tarjan_Split_Components;

const
  Max_Edges = Max_Vertices * 3;

type                          { the following types are necessary to place the     }
                              { data structures onto the heap instead of the stack }
  Triple_Stack_Ptr = ^Triple_Stack;
  Triple_Stack     = Array [1..Max_Vertices] of record   { each triple on the stack has }
                                                 Largest_Element,    { largest DFS number in poss. comp. }
                                                 Split1,             { and the two split vertices }
                                                 Split2  : Vertex_Ptr_Range
                                               end;

  Edge_Stack_Ptr   = ^Edge_Stack;
  Edge_Stack       = Array [1..Max_Edges] of record      { each edge element stores }
                                               Head,
                                               Tail  : Vertex_Ptr;  { start, end vertices of the edge }
                                               Comp  : integer      { and virtual number (0 if graph edge) }
                                             end;
  Edge_Ptr_Range   = 0..Max_Edges;

var
  Old_Component: integer;
  Flag,
  Start_Path   : Boolean;           { if the graph is or is not }
  x, y         : Edge_Ptr_Range;
```

```
  Current      : Vertex_Ptr;        { Current vertex tested     }
  Free_Edge    : 0..Max_Edges;      { first free edge stack posn }
  Free_Triple  : 0..Max_Edges;      { first free triple stack posn }

  Triples      : Triple_Stack_Ptr;  { pointers to the two stacks }
  Old_Edges    : Edge_Stack_Ptr;

{-------------------------------------------------------}
{- Initialisation                                      -}
{-------------------------------------------------------}
procedure Initialise_Hopcroft_Tarjan;

var
  Temp : Vertex_Ptr;

begin
  DFS;                                  { Depth First Search }
  ReOrder_Lists;                        { and Reorder lists - very important }
  Second_DFS (Current);                 { Step 3 of algorithm }

  new (Triples);
  new (Old_Edges);                      { create the two stacks }
  Free_Triple := 1;
  Free_Edge := 1;

  Initialise_Components;                { initialise the data for the components }
end;

{-------------------------------------------------------}
{- Calculates the maximum of two numbers               -}
{-------------------------------------------------------}
function Max (x, y : Vertex_Ptr_Range) : Vertex_Ptr_Range;

begin
  if Num(x) < Num(y)
    then Max := x
    else Max := y
end;

{-------------------------------------------------------}
{- Push a triple (a, b, h) onto the triple stack       -}
{-------------------------------------------------------}
procedure Push_Triple (h, a, b : Vertex_Ptr_Range);

begin
  with Triples^ [Free_Triple] do
    begin
      Largest_Element := h;
      Split1 := a;
      Split2 := b
    end;
  Free_Triple := Free_Triple + 1
end;
```

```
{-------------------------------------------------------------}
{- Pop a triple from the stack if there is one, otherwise   -}
{- return a 0 0 0 triple to mark end of stack.              -}
{-------------------------------------------------------------}
procedure Pop_Triple (var h, a, b : Vertex_Ptr_Range);

begin
  Free_Triple := Free_Triple - 1;
  if Free_Triple > 0
    then with Triples^ [Free_Triple] do
           begin
             h := Largest_Element;
             a := Split1;
             b := Split2
           end
    else begin
      Free_Triple := 1;
      h := 0;
      a := 0;
      b := 0
    end
end;

{-------------------------------------------------------------}
{- Returns the value of the triple top-of-stack. If there is-}
{- nothing on the stack then return 0 0 0 triple.           -}
{-------------------------------------------------------------}
procedure Read_Triple_TOS (var h, a, b : Vertex_Ptr_Range);

begin
  if Free_Triple = 1
    then begin
      h := 0;
      a := 0;
      b := 0
    end
    else
      with Triples^ [Free_Triple - 1] do
        begin
          h := Largest_Element;
          a := Split1;
          b := Split2
        end;
end;
```

```
{-------------------------------------------------------}
{- Push an edge onto the edge stack.                    -}
{-------------------------------------------------------}
procedure Push_Edge (New_Head, New_Tail : Edge_Ptr_Range;
                     Component : integer);
begin
  with Old_Edges^ [Free_Edge] do
    begin
      Head := New_Head;
      Tail := New_Tail;
      Comp := Component
    end;
  Free_Edge := Free_Edge + 1
end;

{-------------------------------------------------------}
{- Pop an edge from the edge stack - if there are none then -}
{- we return the edge  0 0 with virtual number 0.       -}
{-------------------------------------------------------}
     procedure Pop_Edge (var New_Head, New_Tail : Edge_Ptr_Range;
                    var Component : integer);
begin
  if Free_Edge = 1
    then begin
      New_Head := 0;
      New_Tail := 0;
      Component := 0;
    end
    else begin
      Free_Edge := Free_Edge - 1;
      with Old_Edges^ [Free_Edge] do
        begin
          New_Head := Head;
          New_Tail := Tail;
          Component := Comp
        end
    end
end;

{-------------------------------------------------------}
{- Return the edge top-of-stack. If none then return 0 0 0  -}
{-------------------------------------------------------}
procedure Read_Edge_TOS (var x, y : Edge_Ptr_Range);

begin
  if Free_Edge = 1
    then begin
      x := 0;
      y := 0;
    end
    else begin
      x := Old_Edges^ [Free_Edge - 1]. Head;
      y := Old_Edges^ [Free_Edge - 1]. Tail
    end
end;
```

```
{-------------------------------------------------------}
{- The main recursive procedure to perform a DFS and find   -}
{- the separation pairs and their components.               -}
{-------------------------------------------------------}
procedure PathFinder (Start_Vertex : Vertex_Ptr);

var
  Dest_Vertex : Vertex_Ptr_Range;
  Temp_Edge   : Edge_Ptr;

{-------------------------------------------------------}
{- This procedure tests for type 1 split component pairs    -}
{-------------------------------------------------------}
procedure Test_Type_1;

var
  Current_Edge  : Edge_Ptr;
  Old_Component : integer;
  x, y          : Edge_Ptr_Range;

begin
  if (Num(Graph^ [Dest_Vertex]. L2) >= Num(Start_Vertex))     { all edges from dest fall below start_vertex }
                                                          { and either : }
     and ((Num(Graph^ [Dest_Vertex]. L1) <> 1)                { another component above L1 exists }
           or (Num(Graph^ [Start_Vertex]. Father) > 1)        { or another component above Start exists }
           or (Num (Dest_Vertex) > 3))                        { or start_vertex has another edge incident }
    then begin

{ at this stage we have found a split component pair - namely }
{ a = Lowpt1 [Dest_Vertex] and b = Start_Vertex              }

      if debug_tri
        then writeln ('Found a type 1 split - ',
                  Graph^ [Dest_Vertex]. L1:4, Start_Vertex:4);
      Component_No := Component_No + 1;
      Read_Edge_TOS (x, y);                          { output every edge on stack needed }
      while ((Num(Dest_Vertex) <= Num(x))
                and (Num(x) < Num(Dest_Vertex) + Graph^ [Dest_Vertex]. Num_Descend))
         or ((Num(Dest_Vertex) <= Num(y))
                and (Num(y) < Num(Dest_Vertex) + Graph^ [Dest_Vertex]. Num_Descend)) do
        begin    { add all relevant edges to split component }
          Pop_Edge (x, y, Old_Component);
          Add_Comp (x, y, Component_No, Old_Component);    { add to component }
          Graph^ [x]. Degree := Graph^ [x]. Degree - 1;    { decrement degrees }
          Graph^ [y]. Degree := Graph^ [y]. Degree - 1;
          Read_Edge_TOS (x, y);
        end;
      Add_Comp (Start_Vertex, Graph^ [Dest_Vertex]. L1,    { add the virtual edge }
                Component_No, Component_No);
      if Graph^ [Start_Vertex]. A1 >= Num(Dest_Vertex)      { adjust A1 to reflect virtual edge }
        then Graph^ [Start_Vertex]. A1 := Num(Graph^ [Dest_Vertex]. L1);

      { check for multiple edges with virtual edge }

      Read_Edge_TOS (x, y);
      if ((x = Start_Vertex) and (y = Graph^ [Dest_Vertex]. L1))     { check virtual doesnt already }
                                                                     { have multiple twin }
        or ((y = Start_Vertex) and (x = Graph^ [Dest_Vertex]. L1))
```

```
        then begin
          if debug_tri
            then writeln ('Found another Component - multiple edge case');
          Component_No := Component_No + 1;
          Pop_Edge (x, y, Old_Component);      { get old edge }
          Push_Edge (x, y, Component_No);      { store new virtual edge }
          Add_Comp (Start_Vertex,              { add the new component edges }
                    Graph^ [Dest_Vertex]. L1, Component_No,
                    Old_Component);
          Add_Comp (Start_Vertex,
                    Graph^ [Dest_Vertex]. L1, Component_No,
                    Component_No - 1);
          Add_Comp (Start_Vertex,
                    Graph^ [Dest_Vertex]. L1, Component_No,
                    Component_No);
          Graph^ [Start_Vertex]. Degree :=    { decrement degree in graph }
            Graph^ [Start_Vertex]. Degree - 1;
          Graph^ [Graph^ [Dest_Vertex]. L1]. Degree :=
            Graph^ [Graph^ [Dest_Vertex]. L1]. Degree - 1;
        end;

      { now check that no new virtual edges introduced from addition of virtual edge }

      if Graph^ [Start_Vertex]. Father <> Graph^ [Dest_Vertex]. L1
        then begin                                  { inc degrees and add virtual edge }
          Graph^ [Start_Vertex]. Degree :=
             Graph^ [Start_Vertex]. Degree + 1;
          Push_Edge (Start_Vertex, Graph^ [Dest_Vertex]. L1, Component_No);
          Graph^ [Graph^ [Dest_Vertex]. L1]. Degree :=
             Graph^ [Graph^ [Dest_Vertex]. L1]. Degree + 1
        end
        else begin  { another multiple edge with the new virtual edge }
          if debug_tri
            then writeln ('Found another Component - multiple edge case');
          Component_No := Component_No + 1;
          Add_Comp (Start_Vertex, Graph^ [Dest_Vertex]. L1,
                    Component_No, 0);
          Add_Comp (Start_Vertex, Graph^ [Dest_Vertex]. L1,
                    Component_No, Component_No);
          Add_Comp (Start_Vertex, Graph^ [Dest_Vertex]. L1,
                   Component_No, Component_No - 1);
          Current_Edge := Graph^ [Start_Vertex]. Edges;
          while Current_Edge^. Vertex <> Graph^ [Dest_Vertex]. L1 do
            Current_Edge := Current_Edge^. Next;
          Current_Edge^. Virtual_Number := Component_No;    { note virtual edge }
          Current_Edge^. Other_Edge^. Virtual_Number := Component_No
        end
    end
end;
```

```
{-------------------------------------------------------}
{- The type 2 testing                                  -}
{-------------------------------------------------------}
procedure Test_Type_2;

var
  Save_Component,
  Old_Component  : integer;
  x, y, z        : Edge_Ptr_Range;
  Temp_h,
  Temp_a, Temp_b,
  h, a, b        : Vertex_Ptr_Range;

begin
  Read_Triple_TOS (h, a, b);
  while                                          { look at all candidate triples }
        (Num(Start_Vertex) <> 1)    { cannot be at first vertex }
         and (((Graph^ [Dest_Vertex]. Degree = 2)  { easy cond. for degree 2 special }
                  and (Graph^ [Dest_Vertex]. A1 > Num(Dest_Vertex)))  { also must be tree edge }
              or ((h-a-b <> 0) and (Start_Vertex = a))) do    { we are at the start of the triple }

    begin
      if (Start_Vertex = a) and (Graph^ [b]. Father = a)  { not type 2 }
        then begin
          Pop_Triple (Temp_h, Temp_a, Temp_b);
          Read_Triple_TOS (h, a, b)
        end
        else begin
          if (Graph^ [Dest_Vertex]. Degree = 2)   { easy condition! }
             and (Graph^ [Dest_Vertex]. A1 > Num(Dest_Vertex)) { must be tree edge }
            then begin                            { output the triangle }
              Component_No := Component_No + 1;
              Pop_Edge (y, z, Old_Component);                  { remove the first edge }
              Add_Comp (y, z, Component_No, Old_Component);
              Pop_Edge (y, z, Old_Component);                  { remove the second edge }
              Add_Comp (y, z, Component_No, Old_Component);
              Add_Comp (Start_Vertex, z, Component_No, Component_No);  { add virtual edge }
              x := z;                             { note that b = z }
              if debug_tri
                then writeln ('Found a type 2a split - ',
                              Start_Vertex:4, x:4);

              { Add to new component }

              Read_Edge_TOS (y, z);
              if ((y = x) and (z = Start_Vertex))  { check for multiple edges }
                or ((z = x) and (y = Start_Vertex))
                then begin
                  Flag := true;  { signal multiple edge case }
                  Pop_Edge (y, z, Save_Component);   { remove the multiple edge }
                  Read_Edge_TOS (y, z)
                end
            end
            else
              if (Start_Vertex = a)         { candidate separation pair }
                and (a <> Graph^ [b]. Father) { check for condition a > r > b }
                then begin
                  if debug_tri
```

```
                then writeln ('Found a type 2b split - ',
                              a:4, b:4);
              Component_No := Component_No + 1;
              Pop_Triple (Temp_h, Temp_a, Temp_b);
              Read_Edge_TOS (x, y);
              while (Num(a) <= Num(x)) and (Num(x) <= h)
                      and (Num(a) <= Num(y)) and (Num(y) <= h) do
                if ((x = a) and (y = b))  { add the edges to new component }
                   or ((x = b) and (y = a))
                  then begin
                    Flag := true;       { multiple edge case }
                    Pop_Edge (x, y, Save_Component);
                    Read_Edge_TOS(x, y)
                  end
                  else begin  { not a multiple edge }
                    Pop_Edge (x, y, Old_Component);
                    Add_Comp (x, y, Component_No, Old_Component);  { merely add to comp. }
                    Graph^ [x]. Degree := Graph^ [x]. Degree - 1;
                    Graph^ [y]. Degree := Graph^ [y]. Degree - 1;
                    Read_Edge_TOS (x, y)
                  end;
              Add_Comp (a, b, Component_No, Component_No);   { add virtual edge }
              x := b                { note what b is for flag routine }
            end;
        if Flag
          then begin  { at this stage we know a multiple edge exists }
                      { from b to a - variable x stores b            }
            if debug_tri
              then writeln ('Found another Component - multiple edge case');
            Flag := false;
            Component_No := Component_No + 1;  { new multiple edge }
            Add_Comp (x, Start_Vertex, Component_No, Save_Component);
            Add_Comp (x, Start_Vertex, Component_No, Component_No - 1);
            Add_Comp (x, Start_Vertex, Component_No, Component_No);
            Graph^ [x]. Degree := Graph^ [x]. Degree - 1;
            Graph^ [Start_Vertex]. Degree :=
              Graph^ [Start_Vertex]. Degree - 1;
          end;
        Push_Edge (Start_Vertex, x, Component_No);  { note edge is part of this component }
        Graph^ [x]. Degree := Graph^ [x]. Degree + 1;
        Graph^ [Start_Vertex]. Degree :=
          Graph^ [Start_Vertex]. Degree + 1;
        Graph^ [x]. Father := Start_Vertex;  { virtual edge }
        if Graph^ [Start_Vertex]. A1 < Num(x)     { check if first descendant went }
          then Graph^ [Start_Vertex]. A1 := Num(x); { to x, if so then is first descend. }
        Dest_Vertex := x
      end
  end;
end;
```

```
{-------------------------------------------------------------}
{- This procedure is called when we are exploring a path     -}
{- That is, we have not encountered a back edge yet.         -}
{-------------------------------------------------------------}
procedure Explore_Current_Path (Current_Edge : Edge_Ptr);

var
  Head, Tail      : Edge_Ptr_Range;
  First_Edge,
  Finished,
  Deleted_Triple : Boolean;
  Temp_h, Temp_a,
  Temp_b,
  h, a, b,
  x, y            : Vertex_Ptr_Range;
  Old_Component   : integer;

begin
  if Start_Path
    then begin          { we add a special end-of-stack marker }
      y := 0;           { and delete redundant triples         }
      Deleted_Triple := false;
      Read_Triple_TOS (h, a, b);
      if (h - a - b <> 0)       { if not end-of-stack }
        then
          while Num(a) > Num(Graph^ [Dest_Vertex]. L1) do  { cannot be type 2 - a is invalid }
            begin                                          { since L1 implies that we can reach }
                                                           { a part 'higher' in the graph than a }
              y := max (y, h);                      { keep track of max h for a new candidate triple }
              Pop_Triple (Temp_h, Temp_a, Temp_b);  { keep track of Temp_b for lowest part of new triple }
              Read_Triple_TOS (h, a, b);
              Deleted_Triple := true
            end;
      if not Deleted_Triple     { then a candidate is the start and end vertices }
                                { of the current subtree we are about to explore }
        then Push_Triple (Num(Dest_Vertex) + Graph^ [Dest_Vertex]. Num_Descend - 1,
                          Graph^ [Dest_Vertex]. L1, Start_Vertex)
                                { else a candidate is the highest vertex we are about }
                                { to reach and the lowest vertex we have / are reaching }
        else Push_Triple (max (y, Num(Dest_Vertex) + Graph^ [Dest_Vertex]. Num_Descend - 1),
                          Graph^ [Dest_Vertex]. L1, Temp_b);
      Push_Triple (0, 0, 0);   { EOS marker for triple since we start a new subtree }
      Start_Path := false;
      First_Edge := true;      { note that we have started a new path }
    end   { of start path condition }
    else First_Edge := false;

  PathFinder (Dest_Vertex);    { recursively search down the path }
  if (Debug_Tri)
    then write (' Back at ', Start_Vertex, ' ');
  if (Debug_Tri) and (Start_Vertex = 3)
    then write (' Back at ', Start_Vertex, ' ');
  Read_Edge_TOS (Head, Tail);
  Push_Edge (Start_Vertex, Dest_Vertex, Current_Edge^. Virtual_Number);
                           { push the edge as one of future component }
  Test_Type_2;    { test for type 2 splits }

  Test_Type_1;    { test for type 1 splits }
```

```
  if First_Edge   { if we started a path }
    then begin
      Finished := false;
      while not Finished do    { delete off the stack until end-of-stack marker released }
        begin
          Pop_Triple (h, x, y);
          Finished := (h = 0) and (x = 0) and (y = 0)
        end
    end;
  Read_Triple_TOS (h, a, b);
  while (Graph^ [Start_Vertex]. High_Pt > h) and (h-a-b <> 0)
           and (Num(b) > Num(Start_Vertex)) do
    begin     { if highpt is greater then there is an edge into the }
              { 'component' from 'below' the component - so invalid }
      Pop_Triple (h, a, b);
      Read_Triple_TOS (h, a, b)
    end
end;

{-------------------------------------------------------------}
{- We have reached the end of the current path.              -}
{-------------------------------------------------------------}
procedure Reached_End_of_Path (Current_Edge : Edge_Ptr);

var
  Temp_Edge     : Edge_Ptr;
  Old_Component : integer;
  head, tail    : Edge_Ptr_Range;
  y, h,
  a, b          : Vertex_Ptr_Range;

begin
  if Start_Path      { if single back edge }
    then begin
      y := 0;
      Read_Triple_TOS (h, a, b);
      while num(a) > num(Dest_Vertex) do   { delete redundant triples - cannot be type 2 }
        begin                              { since there is an edge from the component out }
          y := max (y, h);
          Pop_Triple (h, a, b);
          Read_Triple_TOS (h, a, b)
        end;
      if y = 0      { then none deleted - new candidate is just the edge }
        then Push_Triple (num(Start_Vertex), Dest_Vertex, Start_Vertex)
                    { else new candidate is from lowest b to new high point }
        else Push_Triple (y, Dest_Vertex, b)
    end;
  if Dest_Vertex = Graph^ [Start_Vertex]. Father  { multiple edge case }
    then begin                                    { so output a triple bond }
      Component_No := Component_No + 1;
      Add_Comp (Start_Vertex, Dest_Vertex, Component_No, 0);     { the two edges }
      Add_Comp (Start_Vertex, Dest_Vertex, Component_No, 0);
      Add_Comp (Start_Vertex, Dest_Vertex, Component_No, Component_No);  { and virtual edge }

      Graph^ [Start_Vertex]. Degree := Graph^ [Start_Vertex]. Degree - 1; { dec degree to show }
      Graph^ [Dest_Vertex]. Degree := Graph^ [Dest_Vertex]. Degree - 1;   { edge is removed }
      Temp_Edge := Graph^ [Dest_Vertex]. Edges;                           { now search for tree edge }
```

```
      while (Temp_Edge^. Vertex <> Start_Vertex) and (Temp_Edge^. Deleted) do
        Temp_Edge := Temp_Edge^. Next;
      Temp_Edge^. Virtual_Number := Component_No;                    { from father to start_vertex }
      Temp_Edge^. Other_Edge^. Virtual_Number := Component_No;       { and mark it virtual         }
    end
    else begin
      Read_Edge_TOS (Head, Tail);
      Push_Edge (Start_Vertex, Dest_Vertex,
                        Current_Edge^. Virtual_Number)   { push the edge }
    end;
  Start_Path := true;
end;

begin    { of pathfinder }
  if Debug_Tri
    then writeln ('Recursing at ', Start_Vertex);
  Graph^ [Start_Vertex]. Used := true;          { we have visited this vertex }
  Temp_Edge := Graph^ [Start_Vertex]. Edges;
  while (Temp_Edge <> Nil) and (Temp_Edge^. Deleted) do  { get first valid edge }
    Temp_Edge := Temp_Edge^. Next;
  While Temp_Edge <> Nil do                              { for every edge from that vertex }
    begin
      Dest_Vertex := Temp_Edge^. Vertex;
      if Num (Dest_Vertex) > Num (Start_Vertex)   { if it is a tree edge }
        then Explore_Current_Path (Temp_Edge)
        else Reached_End_of_Path (Temp_Edge);
      repeat
        Temp_Edge := Temp_Edge^. Next
      until (Temp_Edge = Nil) or (not Temp_Edge^. Deleted);   { and find a new edge to explore }
    end
end;
```

```
{-------------------------------------------------------------}
{- Main Hopcroft and Tarjan procedure                        -}
{-------------------------------------------------------------}
begin
  writeln ('Testing for Planarity');
  Initialise_Hopcroft_Tarjan;         { initialise }
  Flag := false;
  Start_Path := true;
  PathFinder (Current);               { perform the exploration }
  if Debug_Tri
    then begin
      writeln;
      writeln ('Adding the rest of the graph');
      writeln
    end;
  Component_No := Component_No + 1;   { lastly add the remaining edges      }
  while Free_Edge > 1 do              { from the stack to the new component }
      begin
        Pop_Edge (x, y, Old_Component);
        Add_Comp (x, y, Component_No, Old_Component)
      end;
  writeln;
  if Debug_Tri
    then Review_Results (false);
  Merge_Components (false);
  if Debug_Tri
    then Review_Results (false)
end;

end.
```

```
{-------------------------------------------------------}
{- All routines for handling the different components found -}
{- during the separation process, and the merge process.    -}
{-------------------------------------------------------}
unit Component_Handling;

interface

uses
  Planar_Defs;

type
  Component_Types    = (Triple_Bond, Triangle, Other);   { three types of split components }
  Component_Edge_Ptr = ^Component_Edge;
  Component_Edge     = record                            { each component edge has }
                         Head,                             { start vertex }
                         Tail   : Vertex_Ptr_Range;        { end vertex }
                         Origin : Integer;                 { and virtual num (0 if graph edge) }
                         Next   : Component_Edge_Ptr
                       end;
  Components_Array   = Array [1..Max_Vertices] of record     { each component has }
                         Edges         : Component_Edge_Ptr;   { list of associated edges }
                         Comp_Label    : Component_Types;      { the type of the component }
                         Edge_Elements : integer               { the number of edges in the component }
                       end;

var
  Component_No,                          { temporary variable }
  Max_Candidates : integer;              { the number of components }
  Components     : Components_Array;     { the candiadate components }


procedure Add_Comp (x, y, Comp_No : Vertex_Ptr;    { add an edge to a component }
                    Virtual_Number : integer);

procedure Initialise_Components;                   { startup routine }

procedure Review_Results (Use_ST : Boolean);       { review the components found }

procedure Merge_Components (Use_ST : Boolean);     { merge any possible components }


implementation

uses
  Embedd_Globals,
  Planar_Miscellaneous,
  CRT;


{-------------------------------------------------------}
{- An edge is added to the component Comp_No.              -}
{- The edge is (x, y) and is virtual edge no Virtual_Number -}
{- If virtual_number is 0 then the edge is a graph edge.    -}
{-------------------------------------------------------}
procedure Add_Comp (x, y, Comp_No  : Vertex_Ptr;
                    Virtual_Number : integer);
var
```

```
  Temp_Edge_Component : Component_Edge_Ptr;

begin
  if Max_Candidates < Comp_No
    then Max_Candidates := Max_Candidates + 1;    { keep correct Max_Candidates }
  new (Temp_Edge_Component);
  Temp_Edge_Component^. Next := Components [Comp_No]. Edges;
  Components [Comp_No]. Edges := Temp_Edge_Component;
  Components [Comp_No]. Edge_Elements := Components [Comp_No].Edge_Elements + 1;
  with Temp_Edge_Component^ do                    { and add to list element }
    begin
      Head := x;
      Tail := y;
      Origin := Virtual_Number;
    end;
end;

{-----------------------------------------------------------}
{- The global initialisation for the component handling     -}
{-----------------------------------------------------------}
procedure Initialise_Components;

begin
  Max_Candidates := 0;                          { no components yet }
  FillChar (Components, Sizeof(Components), 0);  { defaults all zero }
  Component_No := 0;                            { and last component was 0 }
end;

{-----------------------------------------------------------}
{- The components found are viewed.                        -}
{-----------------------------------------------------------}
procedure Review_Results (Use_ST : Boolean);

const
  Max_Lines = 20;      { max number of lines to be displayed before a prompt }

var
  Out         : text;
  Temp_Edge   : Component_Edge_Ptr;
  Ch          : char;
  Temp_Str    : string;
  Inner,
  Count2,
  Count, Loop : integer;

begin
  TextColor (Red);
  write ('Do you wish to review results? (Y/N) :');    { ascertain interest first ! }
  TextColor (Yellow);
  ch := upcase (ReadKey);
  if ch = #27
    then halt;
  if (Ch = 'Y') or (Ch = 'R')
    then begin
      if Use_St                                 { echo results to a file }
        then Assign (Out, 'Lempel.Cmp')
        else Assign (Out, 'HopTar.Cmp');
      Rewrite (Out);
```

```
      Clrscr;
      writeln ('Edge Head  Edge Tail  Component');
      writeln (Out, 'Edge Head  Edge Tail  Component');
      Count := 2;
      For Loop := 1 to Max_Candidates do              { display each component }
        if Components [Loop]. Edges <> nil
          then begin
            Temp_Edge := Components [Loop]. Edges;    { start edge is first edge }
            Gotoxy (1, Count);
            clreol;
            writeln ('-----------  Component Number ', Loop, '  -----------');
            writeln (Out, '-----------  Component Number ', Loop, '  -----------');
            Count := Count + 1;
            while Temp_Edge <> nil do       { for each edge }
              begin
                Gotoxy (1, Count);
                clreol;
                with Temp_Edge^ do          { write the edge out }
                  begin
                    if not Use_ST
                      then begin
                        write (Out, Head:10, Tail :10);
                        write (Head:10, Tail :10)
                      end
                      else begin
                        write (St_Number_Index [Head]:10,
                                St_Number_Index [Tail]:10);
                        write (Out, St_Number_Index [Head]:10,
                                    St_Number_Index [Tail]:10)
                      end;
                    if Origin = Max_Candidates    { for lempel et al, special case }
                      then Origin := 0;
                    if Origin = 0
                      then begin
                        write ('Graph Edge' : 18);
                        write (Out, 'Graph Edge' : 18)
                      end
                      else begin
                        str (Origin, Temp_Str);
                        write ('Virtual Edge ' + Temp_Str : 18);
                        write (Out, 'Virtual Edge ' + Temp_Str : 18)
                      end;
                    writeln (Out)
                  end;
                Count := Count + 1;
                Temp_Edge := Temp_Edge^. Next;    { get next edge in the component }
                if (count >= Max_Lines)           { check if at bottom of screen   }
                  then begin
                    For Count2 := Count+1 to 24 do
                      begin
                        GotoXY (1, Count2);
                        Clreol
                      end;
                    GotoXY (1, 24);               { prompt user to continue }
                    prompt;
                    Count := 2;                   { and start display from top of screen }
                    Gotoxy (1, 24);
                    clreol
```

```
                end;
              end
          end;                          { finished with current candidate }
      if Count < Max_Lines   { if the final finish is less than the max }
        then
          for Inner := Count to Max_Lines+1 do
           begin
             Gotoxy (1, Inner);
             clreol              { then wipe every line below it }
           end;
      GotoXY (1, 24);
      prompt;           { wait for user to finish }
      Close (Out)
    end;
end;

{-------------------------------------------------------}
{- All components that are of the same type and share a     -}
{- virtual edge are 'merged' by the union of the two less   -}
{- the common virtual edge.                                 -}
{-------------------------------------------------------}
procedure Merge_Components (Use_ST : Boolean);

var
  Temp_Head,
  Temp_Tail        : Vertex_Ptr;
  Success          : Boolean;
  Base_Temp,
  Temp_Dispose,
  Temp_Edge2,
  Temp_Edge        : Component_Edge_Ptr;
  Temp_Flag,
  Merge_Candidate,
  Elements,
  Loop             : integer;

begin
  Elements := 0;
  For Loop := 1 to Max_Candidates do            { first ascertain the type of each component }
    with Components [Loop] do
      begin
        if Edge_Elements = 3       { then it is a bond or triangle }
          then begin
            Temp_Edge := Edges;
            Success := true;
            Temp_Head := Temp_Edge^. Head;      { get the values of first edge }
            Temp_Tail := Temp_Edge^. Tail;
            Temp_Edge := Temp_Edge^. Next;
            while (Temp_Edge <> nil) and (Success) do  { check if the other two edges are the same }
              begin
                Success := Success and
                           ((Temp_Edge^. Head = Temp_Head) and
                             (Temp_Edge^. Tail = Temp_Tail)) or
                           ((Temp_Edge^. Tail = Temp_Head) and
                             (Temp_Edge^. Head= Temp_Tail));
                Temp_Edge := Temp_Edge^. Next
              end;
            if Success
```

```
            then Comp_Label := Triple_Bond      { all edges the same }
            else Comp_Label := Triangle         { edges are different }
          end
          else Comp_Label := Other
      end;
  New (Base_Temp);
  For Loop := 2 to Max_Candidates do            { for every candidate - the first comp. }
                                                { does not share any virtual edges ever }
    with Components [Loop] do
      if Comp_Label <> Other                    { if a bond or triangle }
        then begin
          Base_Temp^. Next := Edges;
          Temp_Edge := Base_Temp;
          while Temp_Edge^. Next <> nil do      { for each edge in the component do }
            begin
              if Temp_Edge^. Next^. Origin <> 0   { if it is a virtual edge }
                then
                  if (Temp_Edge^. Next^. Origin <> Loop) and
                     (Components [Temp_Edge^. Next^. Origin]. Comp_Label = Comp_Label)
                    then begin                  { then if it is shared with another of same type }
                                                { and is not the virtual edge for this component }
                                                { then we have success to merge }
                      writeln ('Merging components ', loop, ' and ', Temp_Edge^. Next^. Origin);
                      Temp_Flag := Temp_Edge^. Next^. Origin;    { the component to merge with }
                      Temp_Dispose := Temp_Edge^. Next;
                      Temp_Edge^. Next := Temp_Edge^. Next^. Next; { wipe the virtual edge }
                      if Temp_Dispose = Edges
                        then Edges := Temp_Edge^. Next;
                      Temp_Edge2 := Temp_Edge;
                      while Temp_Edge2^. Next <> Nil do    { now link with the other component }
                        Temp_Edge2 := Temp_Edge2^. Next;
                      Temp_Edge2^. Next := Components [Temp_Flag]. Edges;   { do the link }
                      while Temp_Edge2^. Next^. Origin <> Temp_Flag do      { search for other virtual edge }
                        Temp_Edge2 := Temp_Edge2^. Next;
                      Temp_Dispose := Temp_Edge2^. Next;
                      Temp_Edge2^. Next := Temp_Edge2^. Next^. Next;        { and wipe it }
                      Components [Loop]. Edges := Base_Temp^. Next;
                      Components [Temp_Flag]. Edges := Nil;
                      Components [Temp_Flag]. Comp_Label := Other;
                      Base_Temp^. Next := nil;                              { finished with this component }
                      Temp_Edge := Base_Temp
                    end
                    else Temp_Edge := Temp_Edge^. Next
                else Temp_Edge := Temp_Edge^. Next;
            end
        end
end;


end.
```

```
{-------------------------------------------------------------}
{- This unit contains global routines used in the main program -}
{- and by the algorithms.                                      -}
{-------------------------------------------------------------}
unit Planar_Miscellaneous;

interface

procedure Prompt;              { waits for a keypress with a message }

procedure GR_Prompt;

procedure Dump_Graph;          { displays the current graph }

procedure Initialise_Graph;    { resets graph values to a empty graph }

procedure Scratch_Graph;       { reclaims memory used by a graph - kills the graph }

procedure Build_Graph;         { reads in a graph from a disk file }

procedure Enter_own_Graph;

Function Check_2_Connected : Boolean;

procedure Generate_Random_Planar_Graph; { generates a random planar graph }

procedure Generate_Random_Non_Planar_Graph; { generates a random planar graph }


{-------------------------------------------------------------}
{-------------------------------------------------------------}
implementation

uses Planar_Defs,      { the type definitions and globals variables }
     Dos,              { Dos is standard TURBOPascal unit }
     CRT;              { CRT is standard TURBOPascal unit }

type
  Face_Range = 1..2000;        { the definitions are private and for random planar graphs }

var
  Face      : Array [Face_Range] of record
                                      Vertex1, Vertex2, Vertex3 : Vertex_Ptr;
                                    end;
  Last_Face : Face_Range;

{-------------------------------------------------------------}
{- Prompts for a key                                          -}
{-------------------------------------------------------------}
procedure prompt;

var Dummy : char;

begin
  while keypressed do
    Dummy := Readkey;
  TextColor (Red);
  writeln ('Press any key to continue...');
```

```
  TextColor (Yellow);
  repeat until keypressed;
  Dummy := ReadKey;
  if Dummy = #27
    then Halt
end;

{-------------------------------------------------------------}
{- Prompts for a key whilst in graphics mode                 -}
{- The only difference with previous is that no message      -}
{-------------------------------------------------------------}
procedure GR_prompt;

var Dummy : char;

begin
  while keypressed do
    Dummy := Readkey;
  repeat until keypressed;
  Dummy := ReadKey;
end;

{-------------------------------------------------------------}
{- The current graph is displayed on the screen              -}
{-------------------------------------------------------------}
procedure Dump_Graph;

var
  Temp_Vertex : Vertex_Ptr;
  Temp_Edge   : Edge_Ptr;
  Scr         : integer;

begin
  clrscr;
  Scr := 0;
  writeln ('The Graph is ');
  writeln;
  writeln ('Physical  Logical   DFS   Degree A1  L1  L2   Edges');
  For Temp_Vertex := 1 to Last_Vertex do  { for every vertex display the edges }
    begin
      write (Temp_Vertex:7, Graph^ [Temp_Vertex]. Number:7);
      write (Graph^ [Temp_Vertex]. St_Number:6,
                  ' ':3, Graph^ [Temp_Vertex]. Degree:5);
      write (Graph^ [Temp_Vertex]. A1:5);
      write (Graph^ [Temp_Vertex]. L1:4);
      write (Graph^ [Temp_Vertex]. L2:4);
      Temp_Edge := Graph^ [Temp_Vertex]. Edges;
      write (' ':8);
      while Temp_Edge <> Nil do
        begin
          write (Temp_Edge^. Vertex:4);
          Temp_Edge := Temp_Edge^. Next
        end;
      writeln;
      Scr := Scr + 1;
      if Scr = 20     { check for full screen and wait if necessary }
        then begin
          Prompt;
```

```
          Scr := 0;
        end;
    end;
  writeln ('and now back to fun! ....');
  writeln;
  prompt;
end;

{-------------------------------------------------------------}
{- Set up an empty graph                                     -}
{-------------------------------------------------------------}
procedure Initialise_Graph;

var Loop : Vertex_Ptr;

begin
  new (Graph);
  For Loop := 1 to Max_Vertices do
    begin
      Graph^ [Loop]. Edges := Nil;
      Graph^ [Loop]. Deleted := false
    end;
  Last_Vertex := 0;
end;

{-------------------------------------------------------------}
{- Wipe the current graph and reset values                   -}
{-------------------------------------------------------------}
procedure Scratch_Graph;

var
  Loop : Vertex_Ptr;
  Temp_Edge,
  Next_Edge : Edge_Ptr;

begin
  if Last_Vertex > 0
    then
      For Loop := 1 to Last_Vertex do
        begin
          Graph^ [Loop]. Edges := Nil;
          Graph^ [Loop]. Deleted := false
        end;
  Last_Vertex := 0;
  File_Loaded := false;
  Global_Filename := ''
end;
```

```
{-------------------------------------------------------}
{- A graph is read in from a file.                      -}
{-                                                      -}
{- The file structure is :                              -}
{-        Every line is of the form                     -}
{-                                                      -}
{- a b b b b b ....                                     -}
{-                                                      -}
{-        That is a single vertex number denoted 'a'    -}
{-              and vertices 'b' adjacent to 'a' follow.-}
{-        Note that if vertices u and v are adjacent, then -}
{-        you must only specify either the edge uv or vu,  -}
{-        but not both. If both are specified, then two    -}
{-        edges will be added to the graph.             -}
{-------------------------------------------------------}
procedure Build_Graph;

const
  StartRow   = 7;            { row to start display of options }

var
  MaxRow,
  MaxCol,
  Row, Col    : Byte;
  FileNames   : Array [1..10, 0..3] of string [12];
  DirInfo     : SearchRec;   { to search through the possible data files }
                             { temporary variables }
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;
  This_Vertex,
  This_Posn,
  Other_Posn,
  Other_Vertex: integer;
  FileName,
  Edge_Set    : string;      { set of edges adjacent from This_Vertex }
  Dummy       : char;
  Inp         : text;        { data file we are reading in from }

{-------------------------------------------------------}
{- This procedure allows the user to interactively enter the-}
{- graph name by panning around a screen of file names.     -}
{-------------------------------------------------------}
procedure Get_Filename;

var
  Current_Char : Char;
  Finished     : Boolean;

begin
  Gotoxy (1, StartRow + MaxRow + 1);
  TextColor (Red);
  writeln ('              Use Arrow Keys.to select a file, press <RETURN> or <Enter> to accept');
  TextColor (Yellow);
  Row := 1;
  Col := 0;
  Finished := false;
  Current_Char := Readkey;
  repeat
```

```
    TextColor (Red);
    GotoXy (1 + 20 * Col, Row + StartRow - 1);    { highlight the current option }
    write (Filenames [Row, Col] : 16);
    Current_Char := Readkey;
    if Current_Char = #0
      then Current_Char := Readkey;
    TextColor (Yellow);
    GotoXy (1 + 20 * Col, Row + StartRow - 1);    { lowlight the current option }
    write (Filenames [Row, Col] : 16);
    TextColor (Black);
    case Current_Char of                          { get movement and move }
      #72 : begin  { up }
              if Row > 1
                then Row := Row - 1
                else Row := MaxRow
            end;
      #80 : begin  { down }
              if Row < MaxRow
                then Row := Row + 1
                else Row := 1
            end;
      #75 : begin  { left }
              if Col > 0
                then Col := Col - 1
                else if Row = MaxRow
                       then Col := MaxCol
                       else Col := 3
            end;
      #77 : begin  { right }
              if ((Row <> MaxRow) and (Col < 3)) or
                 ((Row = MaxRow) and (Col < MaxCol))
                then Col := Col + 1
                else Col := 0
            end;      #79 : begin  { end }
              Col := MaxCol;
              Row := MaxRow
            end;
      #71 : begin  { home }
              Col := 0;
              Row := 1
            e        nd;
      #13 : begin                                 { filename has been selected }
              Filename := Filenames [Row, Col];
              Finished := true
            end
      else begin                                  { error in the input }
        Sound (220);
        Delay (20);
        NoSound
      end
      end
  until Finished;
  GotoXY (1, StartRow + MaxRow + 6);
  TextColor (Yellow)
end;
```

```
begin
  if not Do_Demo                { interactive has a lot more work! }
    then begin                  { set up the screen of available data files }
      clrscr;
      TextColor (Red);
      writeln;
      writeln;
      writeln ('                    The Files in the Data Subdirectory are :');
      TextColor (Yellow);
      writeln ('_____________________________________________________________________________');
      writeln;
      findfirst ('\Data\*.Dat', Archive, Dirinfo);
      Row := StartRow;
      Col := 0;
      while DosError = 0 do
        begin
          Filenames [Row - 6, Col] := DirInfo. Name;
          GotoXy (1 + Col*20, Row);
          write (DirInfo. Name : 16);
          if Col = 3
            then begin
              Col := 0;
              Row := Row + 1
            end
            else Col := Col + 1;
          FindNext (DirInfo)
        end;
      If Col = 0
        then begin
          MaxRow := Row - StartRow;
          MaxCol := 3
        end
        else begin
          MaxRow := Row - StartRow + 1;
          MaxCol := Col - 1
        end;
      GotoXy (1, MaxRow + StartRow);
      writeln;
      writeln;
      writeln ('_____________________________________________________________________________');
      writeln;
      TextColor (Red);
      writeln ('Please enter a filename, or return for Graph.Dat');
      TextColor (Yellow);
      writeln;
      Dummy := readkey;
      if dummy = #0          { user has entered a cursor command so pan around }
        then Get_Filename    { until he/she selects a valid filename           }
        else begin
          if dummy = #13
            then Filename := 'Graph.dat'
            else begin
              write (dummy);
              readln (FileName);
              Filename := Dummy + Filename
            end
        end;
      if Pos ('.', FileName) = 0
```

```
      then FileName := FileName + '.Dat';
  end

  else begin                                          { for a demo mode there is no choice! }
    Filename := 'Draw' + Chr(48+FileNumber) + '.Dat';  { get next file name }
    Filenumber := Filenumber + 1
  end;

Global_Filename := Filename;
FileName := '\Data\' + FileName;
if not Do_Demo
  then writeln ('Reading from ', FileName, '...');
assign (inp, FileName);
{$I-}
Reset (Inp);
{$I+}
if (IOResult <> 0)
  then begin
    File_Loaded := false;
    Global_Filename := 'File Load Error';
    writeln ('File not found error.');
    prompt;
    exit
  end
  else begin
    if not Do_Demo
      then begin
        writeln;
        writeln ('Reading ');
      end;
    File_Loaded := true;
    while not eof(inp) do       { for every vertex in the file }
      begin
        read (inp, This_Vertex);            { read in the vertex to start }
        if not Do_Demo
          then write ('.');
        if Last_Vertex < This_Vertex        { keep track of graph order }
          then Last_Vertex := This_Vertex;
        while not eoln(inp) do              { for every vertex adjacent do }
          begin
            read (inp, Other_Vertex);
            new (Temp_Edge);
            Temp_Edge^. Used := false;      { add to edge list }
            Temp_Edge^. Deleted := false;
            Temp_Edge^. Next := Graph^ [This_Vertex]. Edges;
            Graph^ [This_Vertex]. Edges := Temp_Edge;
            Temp_Edge^. Vertex := Other_Vertex;

            if Last_Vertex < Other_Vertex
              then Last_Vertex := Other_Vertex;
            new (Temp_Edge2);               { and add edge to other vertex's edge list }
            Temp_Edge^. Other_Edge := Temp_Edge2;
            Temp_Edge2^. Other_Edge := Temp_Edge;
            Temp_Edge2^. Used := false;
            Temp_Edge2^. Deleted := false;
            Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
            Graph^ [Other_Vertex]. Edges := Temp_Edge2;
            Temp_Edge2^. Vertex := This_Vertex;
```

```
          end;
        readln (inp)
      end
    end;
  close (inp);              { exit neatly }
  writeln
end;

{-------------------------------------------------------------}
{- The user is prompted to enter his/her own graph for      -}
{- testing.                                                 -}
{-------------------------------------------------------------}
procedure Enter_Own_Graph;

var
  Finished_Line,
  Finished        : Boolean;
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;
  This_Vertex,
  Other_Vertex    : Vertex_Ptr;

{-------------------------------------------------------------}
{- "Fail Safe" routine to get input from the kbd.           -}
{- Fail_Status is true if error                             -}
{-------------------------------------------------------------}
function Get_a_Number (var Fail_Status : Boolean) : Vertex_Ptr;

var
  Temp_Str   : String;
  Temp_Chr   : Char;
  Error,
  Temp_Int   : integer;
  Got_Number : Boolean;

begin
  Temp_Str := '';
  Got_Number := false;
  while (not eoln) and (not Got_Number) do    { get the number charcter at a time}
    begin
      read (Temp_Chr);
      if Temp_Chr in ['0'..'9']
        then Temp_Str := Temp_Str + Temp_Chr   { add to a string storing the number }
        else Got_Number := true
    end;
  if (Temp_Chr = ' ') or (Eoln)               { see if finished nicely }
    then begin
      val (Temp_Str, Temp_Int, Error);        { attempt to get integer within range }
      Fail_Status := (Error <> 0)
                  or (Temp_Int < 1)
                  or (Temp_Int > Max_Vertices);
      if not Fail_Status
        then Get_a_Number := Temp_Int
        else Get_a_Number := 1
    end
    else begin                                { error state }
      Fail_Status := true;
      Get_a_Number := 2
```

```
    end
end;

{-------------------------------------------------------}
{- Main procedure to enter your graph.                 -}
{-------------------------------------------------------}
begin
  Scratch_Graph;
  Initialise_Graph;
  clrscr;
  TextColor (Red);
  writeln;
  writeln;
  writeln ('Please Enter the Graph by giving the adjacency lists of the vertices.');
  Writeln ('First enter the vertex that the adjacency list refers to. ');
  Writeln ('Then give all vertices adjacent to that vertex.');
  writeln;
  Writeln ('For example, if vertex 1 is adjacent to vertices 3, 4 and 5, then you enter');
  Writeln ('1 3 4 5');
  TextColor (Yellow);
  writeln ('_____________________________________________________________________________');
  writeln;

  writeln;
  File_Loaded := true;
  Finished := false;
  while not Finished do                    { for every vertex in the file }
    begin
      Writeln ('Enter a vertex number, followed by the adjacency list');
      Writeln ('or press ''Q'' to quit');
      Writeln ('In both cases please press <Enter> or <Return> when finished.');
      This_Vertex := Get_a_Number (Finished); { read in the vertex to start }
      if not Finished
        then begin
          if Last_Vertex < This_Vertex         { keep track of graph order }
            then Last_Vertex := This_Vertex;
          Finished_Line := false;

          repeat                               { for every vertex adjacent do }
            Other_Vertex := Get_a_Number (Finished_Line);   { get the vertex }
            if not Finished_Line                            { see if finished adj. list }
              then begin
                Temp_Edge := Graph^ [This_Vertex]. Edges;
                while (Temp_Edge <> nil) and (Temp_Edge^. Vertex <> Other_Vertex) do
                  Temp_Edge := Temp_Edge^. Next;     { check for multiple entries }
                if Temp_Edge = nil
                  then begin
                    new (Temp_Edge);
                    Temp_Edge^. Used := false;       { add to edge list }
                    Temp_Edge^. Deleted := false;
                    Temp_Edge^. Next := Graph^ [This_Vertex]. Edges;
                    Graph^ [This_Vertex]..Edges := Temp_Edge;
                    Temp_Edge^. Vertex := Other_Vertex;

                    if Last_Vertex < Other_Vertex
                      then Last_Vertex := Other_Vertex;
                    new (Temp_Edge2);                { and add edge to other vertex's edge list }
                    Temp_Edge^. Other_Edge := Temp_Edge2;
```

```
                    Temp_Edge2^. Other_Edge := Temp_Edge;
                    Temp_Edge2^. Used := false;
                    Temp_Edge2^. Deleted := false;
                    Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
                    Graph^ [Other_Vertex]. Edges := Temp_Edge2;
                    Temp_Edge2^. Vertex := This_Vertex
                  end
              end
          until Finished_Line;
          readln
        end
    end;
  writeln;
  writeln ('Finished Reading in the Graph');
  writeln;
  if Last_Vertex = 0
    then File_Loaded := false;
  prompt
end;

{-------------------------------------------------------------}
{- We check if the graph is 2-connected.                     -}
{-                                                           -}
{- We look at the LowPoint number L1 of each vertex generated -}
{- by the Depth First Search. If L1 = the DFS number of the  -}
{- vertex, then obviously there is no path from this vertex  -}
{- to further up the tree. i.e. the vertex is a cut vertex.  -}
{- For the root we see if the root has two children. If so   -}
{- then the root is a cut-vertex.                            -}
{-                                                           -}
{- See Chapter 1                                             -}
{-------------------------------------------------------------}
Function Check_2_Connected : Boolean;

var
  Count_Kids : integer;
  Loop       : Vertex_Ptr;
  Temp_Edge,
  Temp_Edge2 : Edge_Ptr;

begin
  ClrScr;
  writeln ('Checking that the Graph is 2-connected.');
  if Last_Vertex < 3
    then begin
      writeln ('This graph is not 2-connected');
      Check_2_Connected := false;
      exit
    end;
  For Loop := 1 to Last_Vertex do
    with Graph^ [Loop] do
      begin
        if (L1 = Number) and (Number <> 1)   { check Low Point value }
          then begin
            writeln ('This graph is not 2-connected');
            Check_2_Connected := false;
            exit
          end
```

```
        else if Number = 1        { check for root }
               then begin
                 Temp_Edge := Edges;
                 Count_Kids := 0;
                 while Temp_Edge <list }
            Temp_Edge^. Other_Edge := Temp_Edge2;
            Temp_Edge2^. Other_Edge := Temp_Edge;
            Temp_Edge2^. Used := false;
            Temp_Edge2^. Deleted := false;
            Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
            Graph^ [Other_Vertex]. Edges := Temp_Edge2;
            Temp_Edge2^. Vertex := This_Vertex;
```

```
          end;
        readln (inp)
      end
  end;
 close (inp);              { exit neatly }
 writeln
end;

{-------------------------------------------------------------}
{- The user is prompted to enter his/her own graph for      -}
{- testing.                                                 -}
{-------------------------------------------------------------}
procedure Enter_Own_Graph;

var
  Finished_Line,
  Finished        : Boolean;
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;
  This_Vertex,
  Other_Vertex    : Vertex_Ptr;

{-------------------------------------------------------------}
{- "Fail Safe" routine to get input from the kbd.           -}
{- Fail_Status is true if error                             -}
{-------------------------------------------------------------}
function Get_a_Number (var Fail_Status : Boolean) : Vertex_Ptr;

var
  Temp_Str   : String;
  Temp_Chr   : Char;
  Error,
  Temp_Int   : integer;
  Got_Number : Boolean;

begin
  Temp_Str := '';
  Got_Number := false;
  while (not eoln) and (not Got_Number) do    { get the number charcter at a time}
    begin
      read (Temp_Chr);
      if Temp_Chr in ['0'..'9']
        then Temp_Str := Temp_Str + Temp_Chr   { add to a string storing the number }
        else Got_Number := true
    end;
  if (Temp_Chr = ' ') or (Eoln)               { see if finished nicely }
    then begin
      val (Temp_Str, Temp_Int, Error);        { attempt to get integer within range }
      Fail_Status := (Error <> 0)
                  or (Temp_Int < 1)
                  or (Temp_Int > Max_Vertices);
      if not Fail_Status
        then Get_a_Number := Temp_Int
        else Get_a_Number := 1
    end
    else begin                                 { error state }
      Fail_Status := true;
      Get_a_Number := 2
```

```
    end
end;

{-------------------------------------------------------}
{- Main procedure to enter your graph.                 -}
{-------------------------------------------------------}
begin
  Scratch_Graph;
  Initialise_Graph;
  clrscr;
  TextColor (Red);
  writeln;
  writeln;
  writeln ('Please Enter the Graph by giving the adjacency lists of the vertices.');
  Writeln ('First enter the vertex that the adjacency list refers to. ');
  Writeln ('Then give all vertices adjacent to that vertex.');
  writeln;
  Writeln ('For example, if vertex 1 is adjacent to vertices 3, 4 and 5, then you enter');
  Writeln ('1 3 4 5');
  TextColor (Yellow);
  writeln ('_____________________________________________________________________________');
  writeln;

  writeln;
  File_Loaded := true;
  Finished := false;
  while not Finished do                    { for every vertex in the file }
    begin
      Writeln ('Enter a vertex number, followed by the adjacency list');
      Writeln ('or press ''Q'' to quit');
      Writeln ('In both cases please press <Enter> or <Return> when finished.');
      This_Vertex := Get_a_Number (Finished); { read in the vertex to start }
      if not Finished
        then begin
          if Last_Vertex < This_Vertex         { keep track of graph order }
            then Last_Vertex := This_Vertex;
          Finished_Line := false;

          repeat                               { for every vertex adjacent do }
            Other_Vertex := Get_a_Number (Finished_Line);   { get the vertex }
            if not Finished_Line                            { see if finished adj. list }
              then begin
                Temp_Edge := Graph^ [This_Vertex]. Edges;
                while (Temp_Edge <> nil) and (Temp_Edge^. Vertex <> Other_Vertex) do
                  Temp_Edge := Temp_Edge^. Next;     { check for multiple entries }
                if Temp_Edge = nil
                  then begin
                    new (Temp_Edge);
                    Temp_Edge^. Used := false;       { add to edge list }
                    Temp_Edge^. Deleted := false;
                    Temp_Edge^. Next := Graph^ [This_Vertex]. Edges;
                    Graph^ [This_Vertex]. Edges := Temp_Edge;
                    Temp_Edge^. Vertex := Other_Vertex;

                    if Last_Vertex < Other_Vertex
                      then Last_Vertex := Other_Vertex;
                    new (Temp_Edge2);                { and add edge to other vertex's edge list }
                    Temp_Edge^. Other_Edge := Temp_Edge2;
```

```
                    Temp_Edge2^. Other_Edge := Temp_Edge;
                    Temp_Edge2^. Used := false;
                    Temp_Edge2^. Deleted := false;
                    Temp_Edge2^. Next := Graph^ [Other_Vertex]. Edges;
                    Graph^ [Other_Vertex]. Edges := Temp_Edge2;
                    Temp_Edge2^. Vertex := This_Vertex
                  end
              end
          until Finished_Line;
          readln
        end
    end;
  writeln;
  writeln ('Finished Reading in the Graph');
  writeln;
  if Last_Vertex = 0
    then File_Loaded := false;
  prompt
end;

{-------------------------------------------------------------}
{- We check if the graph is 2-connected.                      -}
{-                                                            -}
{- We look at the LowPoint number L1 of each vertex generated -}
{- by the Depth First Search. If L1 = the DFS number of the   -}
{- vertex, then obviously there is no path from this vertex   -}
{- to further up the tree. i.e. the vertex is a cut vertex.   -}
{- For the root we see if the root has two children. If so    -}
{- then the root is a cut-vertex.                             -}
{-                                                            -}
{- See Chapter 1                                              -}
{-------------------------------------------------------------}
Function Check_2_Connected : Boolean;

var
  Count_Kids : integer;
  Loop       : Vertex_Ptr;
  Temp_Edge,
  Temp_Edge2 : Edge_Ptr;

begin
  ClrScr;
  writeln ('Checking that the Graph is 2-connected.');
  if Last_Vertex < 3
    then begin
      writeln ('This graph is not 2-connected');
      Check_2_Connected := false;
      exit
    end;
  For Loop := 1 to Last_Vertex do
    with Graph^ [Loop] do
      begin
        if (L1 = Number) and (Number <> 1)   { check Low Point value }
          then begin
            writeln ('This graph is not 2-connected');
            Check_2_Connected := false;
            exit
          end
```

```
          else if Number = 1        { check for root }
                 then begin
                   Temp_Edge := Edges;
                   Count_Kids := 0;
                   while Temp_Edge <> nil do
                     begin
                       if not Temp_Edge^. Deleted
                         then Count_Kids := Count_Kids + 1;
                       Temp_Edge := Temp_Edge^. Next
                     end;
                   if Count_Kids > 1
                     then begin
                       writeln ('This graph is not 2-connected');
                       Check_2_Connected := false;
                       exit
                     end
                 end
      end;
  Check_2_Connected := true
end;

{-------------------------------------------------------------}
{- The following Routines are used for the random planar and-}
{- non-planar graphs.                                        -}
{-------------------------------------------------------------}

{-------------------------------------------------------------}
{- An edge is added between vertex1 and vertex2              -}
{-------------------------------------------------------------}
procedure Add_Edge (Vertex1, Vertex2 : Vertex_Ptr);

var
  Temp_Edge2,
  Temp_Edge   : Edge_Ptr;

begin
  new (Temp_Edge);            { create the entries in the adjacency lists }
  new (Temp_Edge2);           { for both of the edges }
  with Temp_Edge^ do
    begin
      Vertex := Vertex2;
      Next := Graph^ [Vertex1]. Edges;
      Other_Edge := Temp_Edge2;
      Deleted := false;              { reset initial fields }
      Used := false;
    end;
  Graph^ [Vertex1]. Edges := Temp_Edge;   { update the first adjacency list }
  with Temp_Edge2^ do
    begin
      Vertex := Vertex1;
      Next := Graph^ [Vertex2]. Edges;
      Other_Edge := Temp_Edge;
      Deleted := false;
      Used := false;
    end;
  Graph^ [Vertex2]. Edges := Temp_Edge2;  { update the second adjacency list }
end;
```

```
{-------------------------------------------------------}
{- A random large graph is generated.                  -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{-------------------------------------------------------}
procedure Generate_Random_Graph (var Last_Face : Face_Range;
                                 Vertices_Placed : Integer);

var
  Loop         : Integer;
  Temp_Edge    : Edge_Ptr;
  Current      : Vertex_Ptr;
  Current_Face : 1..2000;            { a large number of faces }

begin
  Current := Vertices_Placed;
  while Current < Last_Vertex do     { for each new vertex do }
    begin
      Current := Current + 1;
      Current_Face := Random (Last_Face) + 1;      { choose a random face }
      with Face [Current_Face] do
        begin
          Graph^ [Current]. Edges := Nil;
          Add_Edge (Vertex1, Current);             { add edges from new vertex to }
          Add_Edge (Vertex2, Current);             { the three vertices on the face }
          Add_Edge (Vertex3, Current);
          Last_Face := Last_Face + 1;
          Face [Last_Face]. Vertex1 := Vertex2;    { we now have two new faces }
          Face [Last_Face]. Vertex2 := Vertex3;
          Face [Last_Face]. Vertex3 := Current;
          Last_Face := Last_Face + 1;
          Face [Last_Face]. Vertex1 := Vertex1;
          Face [Last_Face]. Vertex2 := Current;
          Face [Last_Face]. Vertex3 := Vertex3;
          Vertex3 := Current;
        end;
    end;
  Last_Vertex := Current;
end;
```

```
{-------------------------------------------------------}
{- A random large maximal planar graph is generated.   -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{- Our initial graph is a triangle.                    -}
{-------------------------------------------------------}
procedure Generate_Random_Planar_Graph;

begin
  randomize;
  File_Loaded := true;
  Scratch_Graph;
  Initialise_graph;
  clrscr;
  writeln ('Please enter the size of the Graph to create');
  readln (Last_Vertex);        { get order of graph }
  writeln;
  Global_Filename := 'Random Planar Graph';
  File_Loaded := true;
  Graph^ [1]. Edges := Nil;
  Graph^ [2]. Edges := Nil;
  Graph^ [3]. Edges := Nil;
  Add_Edge (1, 2);             { form the triangle }
  Add_Edge (1, 3);
  Add_Edge (2, 3);
  Face [1]. Vertex1 := 1;      { note the vertices on the face }
  Face [1]. Vertex2 := 2;
  Face [1]. Vertex3 := 3;
  Last_Face := 1;
  Generate_Random_Graph (Last_Face, 3);   { and build other faces }
end;
```

```
{-------------------------------------------------------}
{- A random large non-planar graph is generated.       -}
{- We store the vertices bounding each face. Then we choose -}
{- a face at random. We insert a new vertex in the face, and-}
{- join all vertices on the face to the new vertex.    -}
{- Note that since we start from a triangle, each face -}
{- always has 3 vertices exactly.                      -}
{- We start with a graph K5 minus one edge. We then generate-}
{- the maximal planar graph using the same method as above. -}
{- Lastly, we add the extra edge.                      -}
{-------------------------------------------------------}
procedure Generate_Random_Non_Planar_Graph;

begin
  Randomize;
  Global_Filename := 'Random Non-Planar Graph';
  File_Loaded := true;
  Scratch_Graph;
  clrscr;
  writeln ('Please enter the size of the Graph to create');
  readln (Last_Vertex);        { get order of graph }
  writeln;
  Graph^ [1]. Edges := Nil;
  Graph^ [2]. Edges := Nil;
  Graph^ [3]. Edges := Nil;
  Graph^ [4]. Edges := Nil;
  Graph^ [5]. Edges := Nil;
  Add_Edge (1, 2);              { we now add all the edges of K5 to the graph }
  Add_Edge (1, 3);
  Add_Edge (1, 4);
  Add_Edge (1, 5);
  Add_Edge (2, 3);
  Add_Edge (2, 4);              { this edge is no added to the face tables below }
  Add_Edge (2, 5);
  Add_Edge (3, 4);
  Add_Edge (3, 5);
  Add_Edge (4, 5);
  Face [1]. Vertex1 := 1;       { Now we note the three vertices bounding each }
  Face [1]. Vertex2 := 2;       { face. We Emphasise that the Edge 2,4 is NOT  }
  Face [1]. Vertex3 := 5;       { added into the face tables. i.e. the face    }
  Face [2]. Vertex1 := 1;       { information relates to K5 minus edge (2,4).  }
  Face [2]. Vertex2 := 5;
  Face [2]. Vertex3 := 4;
  Face [3]. Vertex1 := 3;
  Face [3]. Vertex2 := 4;
  Face [3]. Vertex3 := 5;
  Face [4]. Vertex1 := 2;
  Face [4]. Vertex2 := 3;
  Face [4]. Vertex3 := 5;
  Face [5]. Vertex1 := 1;
  Face [5]. Vertex2 := 3;
  Face [5]. Vertex3 := 4;
  Face [6]. Vertex1 := 1;
  Face [6]. Vertex2 := 2;
  Face [6]. Vertex3 := 3;
  Last_Face := 6;
  Generate_Random_Graph (Last_Face, 5);  { Now generate the appropriate random maximal graph }
end;
```

```
end.
```

```
{-------------------------------------------------------------------}
{- This unit contains global definitions used by all the algorithms -}
{-------------------------------------------------------------------}
unit Planar_Defs;

interface

const
  MaxDemoFiles   = 8;      { The graphs that we are using in the program }
  Debug_Embedding= false;
  Debug_Tri      = false;
  Debug          = false;
  Ordinary       = 0;      { we represent a graph edge as a virtual edge, label 0 }
  Max_Vertices   = 100;    { maximum number of vertices on the graph }

type
  Angle             = real;                    { graph drawing constants }
  Coords            = Integer;
  Vertex_Ptr        = 1..Max_Vertices;         { used to access a vertex }
  Vertex_Ptr_Range  = 0..Max_Vertices;
  Embed_Vertex_Ptr  = 1..Max_Vertices;
  Cycle_Element_Ptr = ^Cycle_Element;          { for the extendible cycles of the drawings }
  Cycle_Element     = record
                        Vertex : Vertex_Ptr;
                        Prev,                  { doubly linked list }
                        Next   : Cycle_Element_Ptr
                      end;
  Cycle             = record
                        Length : Vertex_Ptr_Range;
                        Head   : Cycle_Element_Ptr;
                      end;

  {*************************************************************************}

  PQ_Node_Ptr     = ^PQ_Node;

  {*************************************************************************}
                                     {The General graph structure is next }
  Edge_ptr   = ^edge;                { for each vertex we store a list of edges }
  Edge       = record
                 Virtual_Number : integer;
                 Rank,               { for embeddings - order it appeared in adj}
                 Mark,               { Used in all algorithms for various tasks }
                 Weight : Vertex_Ptr_Range;  { Weighting for edge sortings       }
                 Drawn,              { if edge has been drawn yet               }
                 Deleted,            { we delete edges in Hopcroft, Tarjan      }
                                     { in the DFS routine                       }
                 Used   : boolean;   { Boolean variable to note if the edge     }
                                     { has been tested in the planarity process }
                 Vertex : Vertex_Ptr;  { the vertex this edge is incident to    }
                 PQ_Ptr : PQ_Node_Ptr; { in Lempel et al. - a pointer to the    }
                                     { PQ-tree leaf represented by this edge    }
                 Other_Edge,         { Points to the edge element in the other  }
                                     { vertex's edge list                       }
                 Previous,           { we need doubly linked lists for deletion }
                 Next   : Edge_ptr;  { Next edge element in the edge list       }
               end;
```

```
  vertex     = record                    { each vertex has the following information }
                 Cyc_Clockwise,
                 Cyc_Anticlock,          { clock and anticlock in the region        }
                 Edges    : edge_ptr;    { the edge list of edges incident to vertex }
                                        { Next are for the triconnectivity algorithm }
                 A1       : Vertex_Ptr;  { first element in adjacency list          }
                 Degree,                 { degree of vertex                         }
                 High_Pt,                { as in Ch 3, Sect. 3.2                    }
                 Num_Descend,            { Number of descendants                    }
                 Mark     : integer;     { Used in all algorithms in various tasks  }
                 Father   : Vertex_Ptr_Range; { The Father in the graph for DFS or in }
                                         { Demoucran et al. for fragment generating  }
                 L1,                     { The weighting L1 used in                 }
                                         { Hopcroft & Tarjan and Lempel et al       }
                 L2,                     { The second lowpoint used only in Hopcroft }
                 ST_Number,              { Used in Lempel, Even and Cederbaum only  }
                 Number       : Vertex_Ptr_Range; { depth first number                  }
                 Component_ID : Vertex_Ptr_Range; { to determine the correct cycles }
                 X, Y         : Coords;  { position on screen - only valid if Placed = true }
                 Deleted,                { check by algo for extendible cycle }
                 Placed,                 { placed on the screen             }
                 Queued,                 { about to be placed               }
                 Critical,               { part of a critical pair          }
                 Used     : Boolean;     { if the vertex is used in the current graph}
               end;
  Graph_Ptr  = ^Graph_Structure;
  Graph_Structure = array [Vertex_Ptr] of Vertex;  { the graph is placed on the heap }

  {*****************************************************************************}
                                         { The buckets are used in the       }
                                         { Hopcroft et al and Lempel et al   }
                                         { to perform a linear sort of edges }
  Bucket_Ptr = ^Bucket;
  Bucket     = record
                 Next   : Bucket_Ptr;    { the next edge with the same weighting }
                 Vertex : Vertex_Ptr;    { the vertex this edge comes from       }
                 Data   : Edge_Ptr;      { and the edge element itself           }
               end;

  {*****************************************************************************}
                                         { this section describes the PQ tree }
                                         { data structures used in the        }
                                         { Lempel, Even and Cederbaum alg.    }

  PQ_Type         = (P_Node, Q_Node, Leaf, Indicator);   { Node can be P-node, Q-node, leaf or virtual }
  Pass_Two_Status = (Empty, Full, Partial);             { all possible states }
  Pass_One_Status = (None, Queued, Blocked, UnBlocked); { for the two passes  }
  Double_Ptr      = ^Double_Ptr_Node;
  Double_Ptr_Node = record               { a circular double linked list for  }
                                         { a P-nodes children                 }
                      Left,                 { elements to the left and right  }
                      Right   : Double_Ptr; { of the node in the circular list}
                      Element : PQ_Node_Ptr { and the particular PQ-node child}
                    end;
  List_Ptr        = ^List;               { a singly linked list }
  List            = record
                      Next    : List_Ptr;
                      Element : PQ_Node_Ptr;
```

```
                end;
  PQ_Node    = record
                 Data_Label         : Pass_Two_Status;
                 Full_Kids          : List_Ptr;     { a list of the full children  }
                 Full_Kids_Count    : Integer;      { a count of the full children }
                 Partial_Kids       : List_Ptr;     { partial kids - 0, 1, 2 nodes }
                                                    { never more than 2 nodes      }
                 Immediate_Siblings : List_Ptr;     { siblings - 0, 1, 2 nodes     }
                                                    { zero if the father is P-node }
                                                    { 1 if it is an endmost kid    }
                                                    { 2 otherwise                  }
                 Mark               : Pass_One_Status;
                 Parent             : PQ_Node_Ptr; { parent in the tree           }
                 Pert_Child_Count,                 { number of children pertinent }
                 Pert_Leaf_Count    : Integer;      { number descendant full leaves}
                 Circ_List_Posn     : Double_Ptr;  { points to a P-node parent    }
                 case Node_Type : PQ_Type of       { double circular list posn    }
                   P_Node    : (Child_Count   : Integer;       { number of children  }
                                Origin_Label  : Vertex_Ptr;
                                List_Start    : Double_Ptr);   { and circular list   }
                                                               { of the children     }
                   Q_Node    : (LeftMost_Kid  : PQ_Node_Ptr;   { two endmost children}
                                RightMost_Kid : PQ_Node_Ptr;
                                Joint_Origin  : Vertex_Ptr);
                   Leaf      : (Tail_Vertex   : Vertex_Ptr;
                                From_Vertex   : Vertex_Ptr;
                                From_Origin_Label  : Vertex_Ptr);
                   Indicator : (For_Vertex     : Vertex_Ptr;   { what indicator it is }
                                Left_Sib       : List_Ptr)     { what my "left" sibling is }
                end;
  Embed_Edge_ptr   = ^Embed_Edge;                    { for each vertex we store a list of edges }
  Embed_Edge       = record
                       Next        : Embed_Edge_ptr;    { Next edge element in the edge list       }
                       case Indicator : Boolean of
                         true  : (Indicator_Vertex : Embed_Vertex_Ptr;
                                  Same_as          : Boolean);
                         false : (Tail_Vertex : Embed_Vertex_Ptr);  { the vertex this edge is incident to
                     end;

  Embed_Vertex     = record                          { each vertex has the following information }
                       Used,
                       Reverse : Boolean;
                       Edges   : Embed_Edge_ptr;     { the edge list of edges incident to vertex }
                     end;
  Embed_Graph_Ptr  = ^Embed_Graph_Structure;
  Embed_Graph_Structure = array [Embed_Vertex_Ptr]
                          of Embed_Vertex;  { the graph is placed on the heap }


var
  Is_2_Connected,                  { is the graph 2-connected }
  Do_Demo          : Boolean;      { are we doing the demo }
  FileNumber       : Integer;      { what graph in the demo are we doing }
  Num_Deleted      : Vertex_Ptr_Range;  { how many degree 2 vertices are we deleting }
  Global_Filename  : string;       { name of file we are using - non-demo only }
  Heap_Position    : ^Integer;     { used to release memory after algorithm has ran }
  File_Loaded      : Boolean;      { if a graph is in memory                        }
  Initial_Cycle    : Cycle;        { initial extendible cycle }
```

```
  Graph,
  Graph_G         : Graph_Ptr;       { the actual graph - used for drawing since      }
                                     { Graph.tpu causes conflict with variable called Graph }
  Upward_Embed_Graph,                { upward embedding of the graph }
  Embed_Graph     : Embed_Graph_Ptr; { the embedding of the graph }
  Planar          : Boolean;
  Last_Vertex     : Vertex_ptr_Range;{ the order of the current graph                }


implementation

begin
  Num_Deleted := 0
end.
```