Application of Genetic Algorithms to the Travelling Salesperson Problem

by

Peter John Campbell McKenzie

Dissertation

Submitted in Fulfillment

of the Academic Requirements

for the Degree of

Master of Science in Computer Science,

in the

Department of Computer Science and Information Systems,

University of Natal, Pietermaritzburg

December, 1996

For Lisa

Application of Genetic Algorithms to the Travelling Salesperson Problem

Peter John Campbell McKenzie
University of Natal, Pietermaritzburg, 1996

Supervisor: D. Petkov

Genetic Algorithms (GAs) can be easily applied to many different problems since they make few assumptions about the application domain and perform relatively well. They can also be modified with some success for handling a particular problem. The travelling salesperson problem (TSP) is a famous $\mathcal{NP}$-hard problem in combinatorial optimization. As a result it has no known polynomial time solution. The aim of this dissertation will be to investigate the application of a number of GAs to the TSP. These results will be compared with those of traditional solutions to the TSP and with the results of other applications of the GA to the TSP.

Preface

The experimental work described in this dissertation was carried out in the Department of Computer Science and Information Systems, University of Natal, Pietermaritzburg, from January 1994 to December 1996, under the supervision of Dr Don Petkov.

The studies represent original work by the author and have not otherwise been submitted in any form for any other degree or diploma to any University. Where use has been made of the work of others it is duly acknowledged in the text.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor Dr Don Petkov for his advice, encouragement and guidance in the preparation of this dissertation. I would also like to thank Computer Management for the study leave and the use of computing resources. Finally I would like to thank my wife Lisa for her support and for being such a conscientious proof reader.

PETER JOHN CAMPBELL MCKENZIE

University of Natal, Pietermaritzburg
December 1996

Chapter 1

Introduction

1.1 Background to the Problem Area

Genetic Algorithms (GAs) are an adaptive search technique that form part of the field of evolutionary computing. The standard GA is domain-independent and so is classified as a *weak method* [76] because only the objective function is used to guide the search. It is, however, surprisingly efficient in its application to a wide range of optimization problems. In certain cases, such as the Travelling Salesperson Problem (TSP), domain-specific knowledge can be added to increase efficiency.

The Travelling Salesperson Problem can be simply stated as the problem of finding the shortest route for a salesperson starting at a home city to visit each of a list of cities exactly once and return home. It is an $\mathcal{NP}$-hard problem in graph theory so no 'good' algorithm exists for the TSP. That is, there is no known algorithm that can solve every TSP in polynomial time. There do, however, exist heuristic algorithms that have been shown experimentally to produce near optimal solutions in polynomial time (although theorems exist which show that this behaviour cannot be expected in all cases). The TSP was significant in the development of the theory of computational complexity such as $\mathcal{NP}$-completeness as well as in developments in operations research. Many applications can be formulated as the TSP, or can be converted to a TSP, or contain subproblems that can be solved as TSPs.

One of the benefits of using a GA on any problem is that no matter how

complicated the constraints, a GA can be implemented if an objective function is known. This provides another good reason for studying GA solutions to the TSP — many problems can be phrased as extensions of the TSP which are harder to solve. Given a heuristic procedure to solve the TSP, it is often impossible or difficult to extend it. The GA, however, is more easily extended owing to its domain-independence. Thus the application of a GA to the TSP provides knowledge on the application of the GA to a large class of problems.

Genetic algorithms can be enhanced to use domain-dependent knowledge to improve accuracy and performance. Naturally, if many domain-specific changes have been made in the application of the GA to the TSP, it may not be possible to apply this modified GA to other TSP-related problems.

1.2 Goal and Objectives of the Research

Since the TSP has existing good heuristic solutions (relative to the difficulty of the problem), applying a GA to this problem can have a number of different goals. The goal of this research is to investigate the application of genetic algorithms to the travelling salesperson problem. This goal can be reduced to the following objectives:

- Review the current literature on GAs and the TSP, in particular that which relates to the solution of the TSP using GAs.
- Implement a GA for the solution of the TSP.
- Investigate a number of GA parameters to determine which values produce the best solutions to the TSP. This includes domain-specific modifications to the GA for the TSP to improve performance.
- Compare the empirical performance of the GA developed with the results of other researchers as well as those of the traditional TSP heuristics.

1.3 Scope and Limitations

Firstly, this study is limited to the symmetric TSP where distances between cities are the same in both directions. This is the version of the TSP most often studied. This restriction makes comparison with other heuristic procedures easier. Secondly, the study will compare the GA only with more conventional TSP solutions. No attempt is made to compare results with, for example, the use of neural networks or other specialized techniques to solve the TSP. This is necessary as the TSP must be one of the most widely studied problems in combinatorial optimization and other contemporary approaches to its solution may require separate research projects on their own.

Although GAs are a more recent development than the TSP the field has expanded very quickly. The definition of what constitutes a GA is evolving all the time. This dissertation applies some GA technology that is not part of the classical GA [33] like steady-state generations [25] and hybrid functionality in terms of local search [12] and population seeding [43]. More radical departures from the standard GA model, like CHC [29], were not used.

1.4 Significance

The TSP is a problem of theoretical importance [52, p37] partially due to the effect it had on the development of the field of combinatorial optimization. It is, however, not just of theoretical interest but it or its derivatives also have practical applications [32]. GAs are general search methods so it would be expected that the performance of GAs on the TSP would be too slow to really challenge existing TSP solvers like the Lin-Kernighan heuristic which are really very good given the difficulty of the problem.

The significance of this research is that it demonstrates how a domain-independent method such as GAs can be applied to combinatorial optimization problems such as the TSP. It shows the comparative performance that can be expected and how the GA can be used to improve the results obtained using traditional TSP heuristics. It is also significant because the method can be generalized to more complex and difficult combinatorial optimization problems.

1.5 Research Methodology

A GA was implemented to solve the TSP with a number of variable parameters. Experiments were run with different parameter values to identify the most successful combinations. The GA included hybridization techniques, such as local search and population seeding. Comparisons were also made with a number of results by other researchers and with published results for the Lin-Kernighan heuristic.

There are three different approaches that can be used to compare performance of heuristic algorithms:

- performance guarantees or worst case analysis
- probabilistic or average case analysis
- empirical analysis

In this study only empirical analysis was performed. The difficulty of analysing any good heuristic algorithm for the TSP with methods other than empirical ones is noted by Johnson and Papadimitriou [53, p146] because good heuristics tend to have a great deal of interaction between the different stages that cannot be easily analysed.

1.6 Outline of the Dissertation

The rest of this dissertation is divided into four chapters:

Chapter 2: Genetic Algorithms Here a basic description of the GA is given with some examples. This does not cover any special changes for use with the TSP.

Chapter 3: The Travelling Salesperson Problem The TSP, and traditional heuristic solutions thereof, are described.

Chapter 4: Solving the Travelling Salesperson Problem using Genetic Algorithms This chapter surveys the use to which GAs have already been put in the solution of the TSP. Modifications that have been made to the GA in order to achieve efficient performance are described.

Chapter 5: Experimental Methodology and Results This chapter explains the experiments, including techniques used, and the results of the experiments. The comparative results for these techniques, versus those of traditional GAs and current TSP approaches by others, are analysed.

Chapter 2

Genetic Algorithms

2.1 Introduction

Evolutionary Computing is a wide field that has been studied independently by a number of researchers. Evolutionary Computing originated only in the last thirty years though it uses the ideas of natural selection as expounded by Charles Darwin in the 19th Century. Natural selection is a very powerful method of search, capable of finding viable organisms in a hostile world. This indicates a robust and efficient search method given the relatively short space of time over which complex and diverse life has flourished on this planet.

Evolutionary Computing currently includes the following fields: Genetic Algorithms [33], Evolutionary Programming, Evolution Strategies [3], Classifier Systems [33], and Genetic Programming [57]. The algorithms based on any of these approaches, being motivated originally by evolution, are collectively called Evolutionary Algorithms [48]. This study will look at Genetic Algorithms in particular and will touch on other approaches only where they are relevant.

2.1.1 Evolution

Charles Darwin published *The Origin of Species* in 1859 to explain the theory of evolution. Evolution is often misunderstood as random chance. The truth is that evolution is based primarily on cumulative selection which is very different from random chance

[23]. In cumulative selection, small changes are made by mutation and the best individuals are selected by survival of the fittest. In this way, better solutions are built on good solutions. Random chance is too slow to produce any sort of useful results in the time scales available in this universe. An example that illustrates this well is mentioned in [23, p45]. The haemoglobin molecule in the blood consists of four chains of 146 amino acids. There are 20 different amino acids so there are $20^{146} \approx 8.92 \times 10^{189}$ possible ways in which a chain of 146 amino acids can be assembled. The possibility of building one of these chains by random chance is very small even given the whole life of the universe.

Evolution can be viewed as the search through 'gene space' for good chromosomes. Chromosomes direct the growth of organisms and these chromosomes survive to reproduce only if the organism carrying them survives. Chromosomes can be altered from one generation to the next by the random mutation or recombination of genetic material from parents into a child. As only fit organisms reproduce, only those chromosomes which represent fit individuals will be copied into the next generation. Genetic algorithms mimic the processes of evolution by abstracting out processes of reproduction, recombination and mutation.

One method that was put forward to explain evolution before Darwinism is Lamarckism. Lamarck suggested that physical changes to an individual during its lifetime could be passed on to its offspring. That is, the phenotype could affect the genotype of an individual [23]. This is now known not to happen in nature, but this method has still been successfully applied by some researchers in evolutionary computing [48].

2.1.2 Genetic Algorithm Background

Genetic Algorithms (GAs) are a model of machine learning, and in particular machine search, which is based on the concept of evolution in nature. The basic principles were first described by John Holland in 1975 in the book *Adaptation in Natural and Artificial Systems*. Holland and his colleagues at the University of Michigan are considered the founders of GAs. A good introduction to GAs can be found in the book *Genetic*

Algorithms in Search, Optimization, and Machine Learning by Goldberg, formerly one of Holland's students [33]. This book covers the theory of genetic algorithms, as well as the implementation of a simple GA. A review of GA applications and some more advanced methods that can be applied to GAs are discussed. For Internet resources the *Hitch-Hiker's Guide to Evolutionary Computation* [48] provides valuable pointers for getting started in this field.

The dissertation by De Jong, *An Analysis of the Behaviour of a Class of Genetic Algorithms* in 1975 (reviewed in [33]), provided experimental results on function optimization and made use of Holland's theory. Since then, genetic algorithms have been applied to many different areas, including scheduling [18], multimodal function optimization [37] and building neural networks [89]. They have been converted to work on parallel computers making it possible to attack larger problems [70].

A number of conferences covering GAs have been organized in the past 10 years. In 1985 *The First International Conference on Genetic Algorithms (ICGA)* was held at Carnegie-Mellon University, Pittsburgh [41]. This conference takes place in odd-numbered years [44, 77, 9, 31]. For researchers interested in the theoretical work in GAs, *The first workshop on Foundations of Genetic Algorithms (FOGA)* was arranged in 1990 [73]. It was followed by another in 1992 [92] and another in 1994 [93]. The goals of FOGA include presenting the current state of GA research and providing ideas for future research [92, p1]. The conferences *Parallel Problem Solving from Nature* (PPSN), *Annual Conference on Evolutionary Programming* (EP) and *IEEE Conference on Evolutionary Computation* (ICEC) also include topics on GAs.

The terminology used in the GA literature tends to be a mixture of biological and GA terms. Equivalent GA and biological terms can be found in Table 2.1.

2.2 How Genetic Algorithms Work

While GAs are motivated by biological evolution they do not try to model the process exactly. What is important is that GAs have shown themselves to be useful for search and optimization. This section will describe how a basic GA works. Since there may be some debate over what constitutes a basic GA, the GA described here corresponds

Table 2.1: Corresponding Biological and Genetic Algorithm terms

Biological	**Genetic Algorithm**
gene	feature
allele	feature value
chromosome	string
locus	string position
genotype	structure
phenotype	parameter set or decoded structure

to Goldberg's Simple Genetic Algorithm (SGA) [33]. Suggestions have been made for many extensions and enhancements to the basic algorithm since its inception, some with biological motivation. Changes and potential improvements to the basic GA described here will be explained in Section 2.7.

2.2.1 The Algorithm

Genetic algorithms work on a population of *strings* or *chromosomes* (biological term). The strings encode the parameters of the problem to be solved. An objective function is required to direct the search. The objective function is converted to a *fitness function* which evaluates the fitness of each string[1].

Two *parent* strings are selected for *reproduction* by using the fitness function. Reproduction is performed by means of a *crossover* operator that combines the representations and a *mutation* operator that makes random changes. Each of these operators is applied with some probability, crossover usually having a much higher probability. Since binary strings are often used, the basic *1-point crossover* operator simply selects a crossover point and swaps the strings at that point to produce two new offspring. The *mutation* operator is used to introduce new characteristics not currently present in the population or to reintroduce alleles that have been lost. A common mutation operator changes bits randomly.

[1]Often the fitness function can be used directly as the objective function but sometimes some conversion is needed. For example, fitness functions should be positive.

The application of crossover and mutation to two parent strings produces two *offspring* strings which become part of the next generation. The selection and reproduction process continues until the new generation is full (in other words, the same size as the previous generation). The process now repeats with the new generation. The process is often stopped when a fixed number of generations have been processed. Alternatively some measure of the characteristics of the population, such as no improvement for ten generations, can be used to terminate the GA.

The basic algorithm is listed as Algorithm 1 [7]. The algorithm goes through a number of discrete stages. In the *initialization* phase the initial population is constructed, usually randomly. Then the algorithm enters the *evolution phase.* Here a new population is produced from the old generation. This is repeated until a generation limit is reached or convergence has been obtained. The production of the new generation happens in two steps. One is the *selection* of parents using the fitness function f and the other is the *reproduction* of the offspring from the parents.

Algorithm 1 (The Basic Genetic Algorithm)

Set up an initial population P_0. Let f be the fitness function.

Let $i = 0$.

REPEAT

REPEAT

Let a, b be any elements of P_i with selection biased in favour of fittest individuals as measured by f.

Apply crossover and mutation to a and b to produce two new strings c and d. Insert these into P_{i+1}.

UNTIL $|P_i| = |P_{i+1}|$.

Let $i = i + 1$.

UNTIL P_i has converged or $i >$ maximum iterations.

■

The next subsection discusses the encoding of parameters into strings. After this, the different stages of the GA algorithm just presented, namely initialization, selection, and reproduction, will be described in some detail.

2.2.2 Parameter Encoding

The parameters for the domain to be searched must be encoded into a string which can be manipulated by the GA. The method by which parameters are encoded into the string can make a significant difference to the results. However, since GAs are relatively robust, even less than optimal encodings should produce results, if somewhat inefficiently. The encoded string is called the *genotype* in GAs while what it represents is called the *phenotype*. For example, the encoded set of parameters for the design of an aeroplane wing would be the genotype while the phenotype would be the wing represented by this choice of parameters. The distinction between the genotype and the phenotype is one of the characteristics of GAs. Certain other branches of evolutionary computing, for example, evolution strategies, do not have the genotype/phenotype distinction [3].

Some alphabet has to be used for the strings. Binary strings are most often used. If parameters are encoded into a binary string then the same GA framework, with the crossover and mutation operators described, can be used for many different problems. Also, some GA theory supports using binary strings (see Section 2.5). For the purposes of discussion, the rest of this section will assume that strings use a binary coding. Other representation details will be covered in Section 2.7.

2.2.3 Parent Selection

Parents are selected for reproduction using the fitness function. The process can be divided into two phases. In the first phase each string is assigned a real number, called the *target sampling rate* (TSR) [45] or *expected value* [5], representing how many times this string should ideally be selected for reproduction in the next generation. In the second phase the TSR is converted to an integer using a sampling algorithm so that the actual strings can be selected for reproduction. Each of these two steps is discussed

next.

2.2.3.1 Determining the Target Sampling Rate

In evolution an organism is selected to reproduce on its fitness. The phenotype of the organism is the basis for the survival of the genotype. The same principle exists in GAs where strings are chosen for reproduction by the fitness function, which evaluates the effectiveness of the phenotype represented by a given genotype. In order to use a GA, a fitness function is required for the problem to be solved. No other domain-specific information is required. This function is used to select strings for the next generation population by calculating the target sampling rate for each string.

One of the widely used techniques is *proportional selection* [35]. Here, given a string x, a fitness function f and the current generation P, the probability $p(x)$ that x will be selected once can be calculated as

$$p(x) = \frac{f(x)}{\sum_{z \in P} f(z)}.$$

In a population of size n we let $e(x) = np(x)$, where $e(x)$ is the target sampling rate for x [35]. Thus we have:

$$\begin{aligned} e(x) &= n\frac{f(x)}{\sum_{z \in P} f(z)} \\ &= f(x)\frac{n}{\sum_{z \in P} f(z)} \\ &= \frac{f(x)}{\bar{f}} \end{aligned}$$

where $\bar{f}$ is the average population fitness of the current generation. That is, the target sampling rate is the fitness of the string divided by the average population fitness [45, 33].

2.2.3.2 Selecting a Sample

The target sampling rate gives the number of strings that should go into the next generation. However, it cannot be used directly since the result is a real number, and clearly only a whole number of strings can be selected. There are a number of ways of selecting strings using the fitness function. A common technique is *roulette*

wheel selection, so-called because it uses the same technique as a roulette wheel where each 'slot' in the wheel is a string in the population. Unlike the casino roulette wheel, however, the slots vary in size according to the target sampling rate of the strings they represent. For a population of size n the wheel is spun n times to select the parents that will produce the next generation [33].

A point to note about this form of selection is that theoretically one single string could be selected for every place in the new population[2] [4] — although in practice it is highly unlikely to happen. For this and other reasons there are other selection methods which may be considered superior. These are discussed in Section 2.7.2.

2.2.4 Reproduction Operators

The reproduction operators are used to produce offspring from the parents selected using the fitness function [7]. These operators are not applied unconditionally, but only at some probability. If none of the operators are applied to the parents then the parents are copied unchanged into the next generation. A number of different operators have been proposed but most commonly GAs have two basic operators — crossover and mutation — although the actual form these take may vary. Both of these operators will now be described in more detail.

2.2.4.1 Crossover

Crossover operates on two strings and combines them in some way to produce two new strings. The traditional method is as follows. A point is selected at random in the strings. One new string consists of the first part of the first parent string up to the crossover point followed by the latter part of the second string. The other new string consists of the first part of the second string followed by the latter part of the first string [33]. See Figure 2.1 for an illustration.

By combining two strings in this way it is possible to forge a new string with the best characteristics of the two parent strings. This should hopefully produce a faster search as good *building blocks* from different strings are combined into a single string. Section 2.5.5 will expand on this idea. It is a general premise in GAs

[2]It would need to have a non-zero target sampling rate but that would be the only requirement.

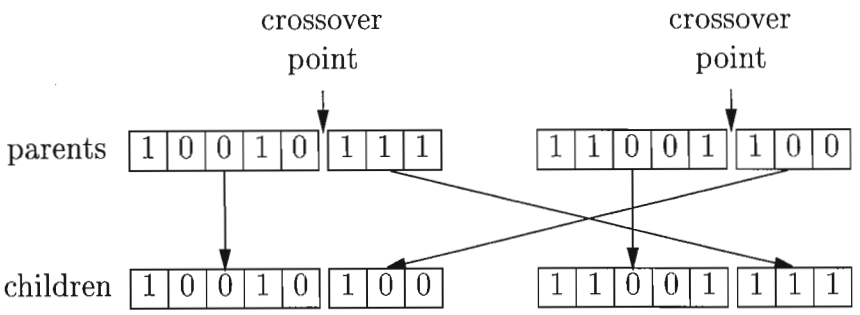


Figure 2.1: Single point crossover operator

that some sort of operator must exist that combines two different strings even if it is not identical to the crossover operator just described [21]. There are other related fields, such as evolutionary algorithms, where this is not the case [48]. Some studies have been inconclusive on the importance of crossover but other studies have shown that crossover does combine building blocks from strings better than mutation [80]. Crossover is normally applied at a higher probability than mutation.

The crossover described here cannot introduce new bit settings if they do not exist in some string that was generated when the population was initialized. For example, if all strings in the population have the first bit set to '1' it will never be possible to produce a string with the first bit set to '0' using the crossover operator described here. This is why a mutation operator is needed.

2.2.4.2 Mutation

The mutation operator adds randomness to the new string. Traditionally, each bit in a selected string is examined and, based on some probability, a decision is made whether to flip that bit. These random changes can introduce new characteristics into the string [33]. See Figure 2.2.

Mutation is credited with a number of useful properties. If allele values are missing from the population, then mutation can introduce new values or reintroduce values that have been lost owing to the 'death' of strings that did not reproduce. Mutation may produce new strings that escape from a local maximum. It also makes small changes in strings, and these changes are useful for local optimization of the solution. Mutation is normally applied at a low rate.

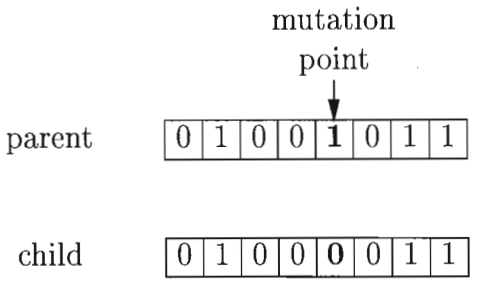


Figure 2.2: Mutation operator

2.3 Other Search and Optimization Methods

In order to understand why GAs have something to offer it is useful to consider other existing optimization techniques to see where GAs differ. Search and optimization is a field with a long history. Many problems have been effectively solved but there are still many problems for which no good solution exists. The techniques given here are general purpose methods that can be applied to a number of domains. This section is based on [7, 33, 76].

2.3.1 Generate-and-Test

Generate-and-test is a simple search strategy. A possible solution is generated, and is tested to see if it meets the required goals [76]. If it does, the process stops, otherwise, another possible solution is generated. This method can be applied in a variety of ways. One extreme is a complete enumeration of the search space. The enumerative method exhaustively searches the whole space. Clearly, infinite spaces have to be discretised first. The disadvantage of this approach is simply one of efficiency. This method performs a thorough search but this may take an impractical amount of time. Some search spaces may be so pathological, however, that this is the only possible method.

Another extreme is a random search. Random walks can never be much better than enumeration on average. Random walks should not be confused with techniques like GAs and simulated annealing that use randomization as part of the search strategy. This technique is not very useful on its own because it does not limit the size of the search space.

Between these extremes is a systematic method that leaves out some parts of the search space because they are not expected to contain good solutions. The parts of the search space to ignore are decided by a heuristic function. Methods like GAs and simulated annealing could be placed in this category.

2.3.2 Hill Climbing

In generate-and-test there is no interaction between the function that generates solutions and the heuristic function. *Hill climbing* is similar to generate-and-test but the heuristic function provides direction for the search [76]. Often the same function that tests if a solution is an acceptable one can be used to direct the next move.

2.3.2.1 Calculus-Based Search

Calculus-based methods make use of the gradient of the objective function. They can be divided into two types. If the gradient can be found analytically it may be possible to solve the set of equations resulting from setting the gradient to zero [72]. This will normally result in a nonlinear set of equations. The problem with this method is that often there is no analytical solution for the gradient of the objective function and the resulting nonlinear equations are themselves difficult to solve. It also has all the disadvantages of the direct calculus-based method below.

The gradient can also be used as a heuristic function to direct a hill climbing algorithm. This is done by beginning the search at some point in the search space and then searching in the direction of steepest ascent. The gradient can be calculated numerically in this case, sometimes making it applicable to a wider range of problems. The disadvantage is that the topology of the space may be nasty so that the search may become misled. Calculus-based methods are also prone to getting stuck in local maxima. Other methods then have to be used to restart the search in other areas of the space.

2.3.3 Combined Heuristic Techniques

Hill climbing methods can be combined with random restarts to explore the whole space better. Once one peak has been located using hill climbing, the algorithm is restarted at a random location [33]. This can help to deal with multimodal search spaces where more than one answer is required. It has the disadvantage that no knowledge from a particular hill climbing phase is carried over into the next climb so progress is fairly linear.

2.3.4 Simulated Annealing

Simulated annealing is modeled on the cooling of material such as metal to a solid state. Kirkpatrick *et al* demonstrated the application of these techniques to combinatorial optimization problems in 1983 [22]. If cooled quickly from a liquid to a solid state, a metal will solidify at a high energy state. If cooled more slowly a lower state of energy will be achieved, but cooling too slowly will waste time. When physical substances are cooled they naturally move toward a lower energy configuration but there is a probability $p = \exp^{-\Delta E/kT}$ that a transition to a higher energy state will occur where ΔE is the change in energy level, T is temperature and k is Boltzmann's constant [76].

The physical process of annealing can be adapted into a form of hill climbing where there is a probability of accepting bad moves while the temperature is high which drops as the temperature is lowered. Assume we have an objective function f and a random initial point x is selected. The algorithm now consists of a move made in any random direction δ. x is then set equal to $x + \delta$ if $f(x + \delta)$ represents an improvement over $f(x)$. If it does not, it may still be accepted with probability $p(t)$ where t is time and p is a monotonically decreasing function. This process repeats until a time limit is reached or convergence is obtained. Thus the algorithm starts off as a fairly random walk but becomes progressively more like hill climbing as time progresses [7]. Unlike hill climbing, however, the result is achieved without using a heuristic function based on the objective function to generate the next move.

For annealing to work well, a good *annealing schedule* must be designed that gives the rate at which the system will cool. Cooling too quickly gives less than optimal

results while cooling too slowly is time consuming. The design of a good annealing schedule must in general be done empirically since it is problem-dependent [22].

2.3.5 How Genetic Algorithms Differ from Other Search Methods

One can see that GAs are a form of generate-and-test but not of hill-climbing because an objective function (or heuristic based on the objective function) is not used directly to generate a solution. They are also similar to simulated annealing because of the use of mutation and randomness to select parents. A major difference is that GAs use a population of points when searching a space, giving a more overall view of the search space. A population of points enables GAs to find and concentrate on areas of high fitness while still retaining some points of lower fitness, making it easier to avoid local maxima traps. Another major difference is how the parameters are encoded into a string, making it possible, if the string encoding is effective, for genetic operators to recombine the best features of two individuals into a new offspring and so fulfill the *building block hypothesis* [33, p41]. This process will be described more fully in Section 2.5.5.

Genetic algorithms provide a balance between exploration and exploitation [33, p36]. While the selection of strings for the next generation is biased by fitness, it is still possible for less fit strings to make it into the new population. The presence of sub-optimal strings in the population results in exploration which can lead to new high-performance areas of the search space being located. The disadvantage of continued exploration is that GAs are slower than, for example, hill-climbing on easy problem domains.

Other techniques like simulated annealing also provide a balance between exploration and exploitation but, unlike GAs, simulated annealing does not deal with more than one point. In contrast, a well-designed GA can infer the shape of the search space to make better use of the exploration information.

2.4 Genetic Algorithm Applications

Genetic algorithms can be applied to a very wide range of problems because they don't rely on a particular problem structure or domain-specific knowledge. Only an objective function and a method of encoding the problem into a string is required. To improve performance it may be required that more work is done but as a first attempt this is all that is required. Applications include many combinatorial optimization problems in graph theory, engineering control, engineering design, pattern recognition, and economics. A short description of some applications to which the GA has been put will be given in this section to illustrate the possibilities without claiming to be complete.

Function optimization Genetic algorithms were applied to function optimization very early in their development by De Jong. A variety of different functions were tested including those which are difficult for traditional methods, including discontinuous, multimodal and stochastic functions [33].

Optimization of pipeline systems Goldberg's Ph.D. studies were on the optimization problems in gas pipeline control. This problem can be modelled as a set of nonlinear state transition equations where the objective is to minimize the power required to pump the fuel through a pipeline with a number of compressors while meeting given pressure constraints. There is no analytical solution to the problem even for simple cases [33].

Medical image registration Fitzpatrick, Grefenstette and Van Gucht used GAs to align two x-ray images of an artery before and after injection with dye. Once this has been done it is possible to subtract one image from the other, leaving the dye-coated artery wall visible [33].

Job shop scheduling The job shop problem (JSP) is an $\mathcal{NP}$-hard problem which is in practice actually harder than the TSP. N jobs have to be scheduled, using M machines, such that the elapsed time is minimized. Results obtained have been comparable to those using branch and bound [18, 68].

Database query optimization Genetic algorithms have been used to optimize database queries in relational databases. Given an input query the query optimizer must select a search strategy that has minimum cost in terms of CPU and I/O time required to service the query. Results have shown that GA-based systems can out-perform conventional query optimizers on queries with many relations [10].

Face identification A system that evolves faces based on human interaction has been developed to produce identikits of criminal suspects. Face characteristics are encoded in a string so that normal GA techniques of crossover and mutation can be used. An eye-witness is asked to rate 20 randomly generated faces. This rating is used as a fitness function in the evolution of the next generation of 20 faces. The process continues until an acceptable picture of the suspect is evolved [13].

Partitioning problems N objects have to be partitioned into K groups such that some objective function is optimized. There are K^N ways in which this can be done. For some objective functions the partitioning problem is $\mathcal{NP}$-hard. For example, if the N objects are vertices in a planar graph, $K = 4$, and the objective is to minimize the number of adjacent vertices in the same partition, then this is an $\mathcal{NP}$-hard *graph colouring problem* [54].

Timetabling The problem of scheduling classes, teachers and rooms into fixed times so that there are no resource clashes has been solved using a GA. As with most GA problems it is possible to tackle this problem using parallel computing [1].

This selection of applications demonstrates the success which GAs have had with a variety of applications in practice. A lot of work has been done to explain some reasons for their success. This not only makes one feel confident to apply a GA to a particular problem but can be used to guide the choices that must be made when working on an application. Choice of encoding and the definition of the fitness function can be partially based on GA theory. The following section will provide a look at some of this GA theory.

2.5 Some Genetic Algorithm Theory

Genetic algorithms search efficiently because they effectively search a number of hyperplanes in parallel owing to the interaction of selection and recombination. This process has been described as *implicit parallelism* by Holland [33]. Goldberg mentions a number of theories that explain some of the performance advantages that implicit parallelism gives GAs over other search methods. These include the fundamental theorem of genetic algorithms, the k-armed bandit problem, effective schema processing and the building block hypothesis [33]. Each will now be described in some detail.

2.5.1 Schemata Theory

A *schema* (plural schemata) or *similarity template* is a way of describing a set of strings with certain symbols at given string positions. It was developed by Holland. It is useful in describing some of the theory behind the success of GAs. This description is based on [33, p19][45].

If we assume we are dealing with binary strings then a schema consists of strings from the alphabet $\{0, 1, *\}$ where the '$*$' symbol is a wild card meaning either '1' or '0'[3]. So the schema $011*1*0$ represents the set of strings $\{0110100, 0110110, 0111100, 0111110\}$. Conversely one can say that the string 0110 contains the schemata $*11*$, $0*10$ and $0**0$, among others. We now define a number of terms relating to schemata. The *defining length* of the schema is the distance between the first fixed symbol and the last fixed symbol. For a schema H the defining length of H is denoted $\delta(H)$. The defining length of $**1*0*$ is two. The *order* of the schema is the number of fixed positions in the schema, i.e. the number of non-'$*$' symbols. For a schema H the order of H is denoted $o(H)$.

It is interesting to note that for strings of length l the number of schemata is 3^l since for each digit in the string we have three choices. This is in contrast to the number of strings of length l which is 2^l. This assumes, as stated earlier, that we are dealing with binary alphabets. In general, if the encoding alphabet has k symbols then for strings of length l we have $(k+1)^l$ schemata.

[3]It is also common in the literature to use a '#' symbol as a don't care symbol[7, 45].

Now consider the schemata contained in a given string. If we have a string s of length l then the string contains 2^l schemata since each position in the string can take on its value or the '$*$' symbol. So for a population of size n we have between 2^l and $2^l n$ different schemata contained in the population depending on the diversity of the strings. Now consider the GA operating on the schemata rather than on the binary strings. How many schemata survive into the next generation? Clearly if a string contains schemata that are fit (i.e. their presence in the string makes the string fit) it has a high chance of reproducing. Also, if a schema is of a short defining length it will have a lower chance of being disrupted by the crossover operation. Mutation at low levels also has a small chance of disrupting short defining length schemata. This suggests that fit, short defining length schemata have a good chance of being in the next generation. A more theoretical explanation follows to back up these ideas.

2.5.1.1 Hyperplanes

Schemata can also usefully be thought of as hyperplanes. That is, for a schema of length l we may consider a space of dimension l where the schemata represent planes in the space. For example if we consider $l = 3$ (three-dimensional space) then schemata of order three are points, schemata of order two are lines, schemata of order one are planes and the schema of order zero represents the whole space. When considered in this way, a very good graphical picture can be gained of the manner in which GAs search a given space. In the literature the term hyperplane is sometimes used in preference to schema [45, 64].

2.5.2 Exponential Trials to Fit Schemata

Schemata are a useful tool for analysing the performance of GAs. One interesting aspect to examine is how fit schemata reproduce during a GA run. Let $m(H, t)$ represent the number of schemata H in the population at time t. Assuming reproduction in a population of size n with no mutation or crossover we have:

$$m(H, t+1) = m(H, t) n f(H) / \sum_{j=1}^{n} f_j$$

where $f(H)$ is defined as the average fitness of all the strings containing H at time t [33, p28]. Since $\bar{f} = \sum_{j=1}^{n} f_j/n$ is the average fitness we may rewrite this equation as:

$$m(H, t+1) = m(H, t) f(H)/\bar{f}. \tag{2.1}$$

Equation 2.1 shows that the number of occurrences of a particular schema, H, in the population grows at a rate proportional to the ratio of the average fitness of occurrences of H in the population to the average fitness of the entire population. If it is assumed that $f(H)$ is always above average, say $f(H) = (1+c)\bar{f}$ where c is a constant, then for $t = 0$ we have $m(H, 1) = m(H, 0)(1+c)$, so $m(H, 2) = m(H, 0)(1+c)^2$. Thus

$$m(H, t) = m(H, 0)(1+c)^t. \tag{2.2}$$

Equation 2.2 demonstrates that consistently superior schemata will be selected at an exponentially increasing rate. What must still be considered are the effects of crossover and mutation.

Under 1-point crossover a schema always survives when the crossover point does not divide any of the fixed positions in the schema from each other. There is still a possibility that the schema will survive but the lower bound will be considered here. Assuming 1-point crossover the probability p_d that a particular schema H is destroyed is $p_d \leq \delta(H)/(l-1)$ because the schema may be destroyed whenever any of the $\delta(H)$ sites in $(l-1)$ is selected for crossover. Thus the probability of survival p_s is given by $p_s \geq 1 - \delta(H)/(l-1)$. Let p_c be the probability of the crossover being performed. We then have that

$$p_s \leq 1 - p_c \frac{\delta(H)}{(l-1)}. \tag{2.3}$$

Combining Equation 2.1 and Equation 2.3 gives

$$m(H, t+1) \geq m(H, t) f(H)/\bar{f} \left[1 - p_c \frac{\delta(H)}{(l-1)}\right]. \tag{2.4}$$

Equation 2.4 demonstrates that for schemata with short defining lengths the observations from Equation 2.2 will still hold. The only further effect to take into account is that of mutation.

Let p_m be the probability of random mutation of each allele value. Then each allele value has a $1 - p_m$ chance of survival. Since H has $o(H)$ fixed positions, the

probability of survival of the whole of H is $(1-p_m)^{o(H)}$. For small p_m the products of p_m can be ignored so the probability of survival of H is $1-p_m o(H)$. Introducing the effect of mutation into Equation 2.4 gives

$$m(H,t+1) \geq m(H,t)f(H)/\bar{f}\left[1-p_c\frac{\delta(H)}{(l-1)}-p_m o(H)\right] \tag{2.5}$$

if small products are ignored. Equation 2.5 demonstrates that for low mutation, and short defining length schemata with above average fitness, exponentially increased selection will be allocated to these schemata. This result is known as the *Fundamental Theorem of Genetic Algorithms* or the *Schema Theorem* and is significant because when dealing with an uncertain situation, the GA allocation of trials, as described by the Fundamental Theorem, is the correct strategy to balance exploration and exploitation. In the next section, the k-armed bandit problem is examined to explain why the GA strategy is a good one.

2.5.3 The k-Armed Bandit Problem

The 2-armed bandit problem is a problem in decision theory. A 2-armed bandit has one arm that produces award μ_1 with variance σ_1 and a second arm that produces award μ_2 with variance σ_2. The problem is to design a strategy that will produce the best payoff. This problem requires a balance between finding which arm has the best payoff and exploiting that knowledge. It has been shown that the GA strategy is similar to the optimal solution. The difference is that the GA operates on a number of k-armed bandit problems at the same time [33, p36].

2.5.4 Effective Schema Processing

Holland has calculated the effective schema processing as $O(n^3)$ [33]. That is, when a GA processes n strings, effectively n^3 schemata are processed. This result, however, should be considered in the context of the assumptions made to derive it. The following explanation is based on [33, p40].

Let P be a population of n strings of length l. We would like to consider schemata of length l_s that will survive crossover and mutation with probability p_s, a

constant. Now the probability of destruction by 1-point crossover is $p_d = (l_s+1)/(l-1)$. So we want $p_d \leq 1 - p_s$. Select ε such that $\varepsilon < 1 - ps$. Then $p_d < \varepsilon$ will meet the requirements that schemata will survive crossover with probability p_s. We assume that the effect of mutation can be ignored owing to low mutation rates. Thus we have $\frac{l_s-1}{l-1} < \varepsilon$ that is, $l_s < \varepsilon(l-1)+1$.

Now a schema of length l_s can occupy $l - l_s + 1$ different positions in a single string in P. Each l_s size section in a single string can match 2^{l_s} different schemata because each position can either be the '$*$' or match the fixed symbol in that position. At least one position must be the fixed symbol so that the trivial schema of all '$*$' is ignored. Thus there are 2^{l_s-1} options. Multiplying we get

$$2^{l_s-1}(l - l_s + 1).$$

This is the number of schemata in a single string of length l_s. Multiplying by n gives

$$n2^{l_s-1}(l - l_s + 1).$$

which is an upper bound on the number of schemata in the population because there will be duplicates in different strings. To calculate a lower bound let $n = 2^{l_s/2}$. Consider the number of schemata of order $l_s/2$. There are $2^{l_s/2}$ different schemata of order $l_s/2$ so the population should contain at most one schema of order $l_s/2$ or higher. Since the number of schemata of each size is binomially distributed, half of the schemata are larger than order $l_s/2$ and half are smaller [33, p40]. If only the higher order are considered then

$$n_s \geq n2^{l_s-2}(l - l_s + 1)$$

where n_s is the number of schemata processed. Since $n = 2^{l_s/2}$ we have that

$$n_s \geq \frac{n^3(l - l_s + 1)}{4}.$$

But

$$n_s \leq \frac{n^3(l - l_s + 1)}{2}$$

thus $n_s = Cn^3$, that is the number of schemata processed is $O(n^3)$.

The above result is interesting and impressive but as the proof relies on the population being evenly distributed it accurately applies only to the first generation

[45]. However, it does give some idea of the performance that a GA can deliver, especially in the first few generations. It should also be noted that this proof uses a relationship between string length and population size.

2.5.5 Building Block Hypothesis

Effective processing of schemata is interesting but unless their recombination by crossover produces even better strings the performance achieved by GAs will only be slow and linear. The *building block hypothesis* suggests otherwise [33]. This suggests that GAs work by forming highly fit schemata with a short defining length. These *building blocks* are then combined by crossover to form even better building blocks which in turn can be used in the next generation. The building blocks must have a short defining length or they will have a high probability of being disrupted by crossover as has been discussed.

In well-designed GAs, where the hypothesis holds true, very good performance can be achieved. By designing encodings and crossover schemes intelligently the chances of the hypothesis being true in a given GA are increased.

2.6 Problems With Genetic Algorithms

Genetic algorithms as described in this chapter suffer from a number of problems when applied to some problem areas. In this section a number of the commonly recognized problems will be discussed. Some of the methods in Section 2.7 can be used to remove or reduce the effect of these problems.

2.6.1 Epistasis

In some chromosomes the fitness of one gene will be dependent on the fitness of another gene elsewhere in the chromosome. Geneticists call this *epistasis* in particular when it refers to some sort of masking effect. In evolutionary computing epistasis is used more generally to indicate strong gene interaction [8]. If a problem has high epistasis then the building block hypothesis fails because the recombination of two fit chromosomes

often produces a chromosome that is unfit. Epistasis can be reduced by careful design of chromosome representations. It has been shown that any problem can be coded so as to remove epistasis in many cases but finding this coding may be so difficult that it is not practical [88].

2.6.2 Genetic Drift

Genetic drift is the term used to describe the process in which populations converge to a single allele value. This happens even if there is no difference in the selection pressure of the alternative allele values. Genetic drift occurs because of errors in selection that accumulate when dealing with finite populations. The same effects have been seen by geneticists [38]. When a GA is applied to a multimodal function, genetic drift will result in the population converging to a single point [37]. This means that without modifications, GAs cannot solve multimodal problem spaces where more than one result is required. With the help of other techniques it is possible to overcome this in some problem areas [50].

2.6.3 Premature Convergence and Slow Convergence

In *premature convergence* the whole population becomes very uniform, that is it converges, but this resulting population does not contain optimal or near-optimal structures. This can be caused by a number of factors, including too small a population and genetic drift [37]. The opposite problem can also occur where the population does not exploit good individuals and so wastes time exploring the search space when it should make use of available resources. This is known as *slow convergence*. It may be possible to avoid premature convergence or slow convergence using scaling and ranking (see Section 2.7.7).

2.7 Improvements to the Basic Genetic Algorithm

This section looks at changes that can be made to the basic GA described so far. We would like these changes to improve the GA. There are at least three measures to be

considered:

- How good are the results produced?
- How fast were the results produced?
- How much variance is there between GA runs?

If a change improves all these measures on a particular problem, then it is definitely a good change for that problem. However, depending on our goals, some changes which adjust the balance between these measures may also be worth including in a GA. The problem domain is also an issue. Some of the changes may make the GA less robust. Other changes may work well in one problem domain but badly in another.

It is useful to realise that the basic GA is not designed originally as an optimizer [24]. Holland characterised the behaviour as assigning a finite number of trials in an optimal manner so as to balance the requirements for exploration and exploitation in an uncertain environment. The requirements for a function optimizer are different and so some changes are necessary to make a GA-based optimizer really efficient. However, once some of these changes have been made, the theoretical results in Section 2.5 may no longer apply. Some of the changes in this section fall into that category.

Areas in which improvements can be considered:

Operator Probabilities How should crossover and mutation probabilities be calculated? Fixed default values would not seem ideal for all problems. Adjusting the probabilities during a run is also possible.

String Encoding Only binary coding has been discussed. What other methods could be used for coding parameters?

Sampling Algorithms The conversion of the target sampling rate into an actual sample from the population has been mentioned as an area where errors can happen. How can these be reduced?

Scaling and Ranking Often the objective function is used directly as a fitness function but the problems this can lead to include super individuals dominating the

population and slow convergence at the end of the GA run. Scaling and ranking can be used to reduce some of these problems.

Crossover Operators Only a simple 1-point crossover has been described. How else can crossover be implemented?

Hybrid Genetic Algorithms Often GAs are applied to problems which already have a number of known good techniques. How can these be incorporated into a GA?

2.7.1 String Encoding Methods

When performing function optimization one of the problems is that standard binary codings have a performance problem because some consecutive numbers have very different representations. For example, considering numbers represented in four bits, the number 7 is represented by 0111 while 8 is represented by 1000. So the transition from 7 to 8 requires all four bits to change. This is known as a *Hamming cliff* [65, p1]. The mutation operator performs local optimization but has a very small chance of crossing a Hamming cliff.

2.7.1.1 Gray Codes

One method that can be used to solve the Hamming cliff problem is *Gray codes.* Gray code is a function G that forms a one-to-one mapping between the integers $0 \leq i \leq 2^N - 1$ for some $N \geq 0$ [72]. The interesting fact about this coding is that the binary representations of adjacent integers, when converted to Gray code, differ in only one bit. That is, $G(i)$ and $G(i+1)$ differ in only one bit for all integers $i \geq 0$. Gray code is not unique but a commonly used code gives the sequence $(000, 001, 011, 010, 110, 111, 101, 100)$ for 0–7. For this particular coding $G(i) = i \text{ XOR } \lfloor i/2 \rfloor$, where XOR is the bitwise exclusive-or operator [72]. It has been shown that Gray coding can improve the performance of GAs in some cases [65].

2.7.1.2 Real Parameter Genetic Algorithms

One argument for the use of binary for codings rather than some other alphabet is because binary strings contain more schemata than strings coded using larger alphabets. Recall that, for a string of length l on an alphabet of cardinality k there are k^l different strings and $(k+1)^l$ different schemata. So the binary coding maximizes the number of schemata [33, 80]. However, other interpretations of schemata have contradicted this view, suggesting that there are more schemata in non-binary alphabets [94].

Using natural alphabets can have other advantages over using binary representations because more natural operators can be used. For example, crossover can be defined to be the average of two parents and mutation some random creep obtained by adding or subtracting a small random amount [8]. It is these advantages — being able to use problem dependent operators — that some authors argue gives real parameter GAs an advantage. This has been backed up by some experimental results [51].

2.7.2 Sampling Algorithms for Parent Selection

As has been explained before, the fitness function is used to calculate the target sampling rate for each string, which is the theoretical number that should go forward for reproduction. Since this number is a real number a method is needed to convert it to an integer value. A sampling technique is used to select the actual parents that will produce new offspring. The success of a given sampling technique can be characterised by the following measures [5]:

bias The bias of a particular method is the difference between a string's expected sampling probability and the actual sampling probability. This value should be as low as possible. It is possible to achieve zero bias.

spread The spread represents the variation possible in each sampling cycle from the optimal. Since integral values must be used in the actual sample, there will always be some spread but it should be kept small. The *minimum spread* possible for a string s in one generation is $\lfloor e(s) \rfloor, \lceil e(s) \rceil$ where $e(s)$ is the expected number of strings s to be selected.

efficiency Any algorithm should be efficient. Since GAs have a high efficiency ($O(nl)$ where n is population size and l string length) the sampling algorithm should not be so expensive as to increase the GA's time complexity.

The only selection technique described so far is roulette wheel selection, which is also called *stochastic with replacement* [5][4]. This method has zero bias since the probability of selection of a string exactly matches the probability represented by the fitness of the string. The spread is, however, unlimited because the actual number of strings selected for a particular individual can vary between zero and the population size. A further problem with this method is that it is not very efficient. It has efficiency $O(n \log n)$ for population size n and that is if the method is implemented using binary search trees. Simple implementations which make multiple passes through the population have efficiency $O(n^2)$ [5].

A number of techniques can be used to reduce the spread. One method is to decrease the expected value of a string each time it is selected (if it moves below zero it is set to zero). This puts an upper bound on the number of times a string can be selected but it increases the bias of the sample and provides no lower bound on the spread.

The other primary technique is to sample the integral and remainder portions of the expected value independently. These techniques are called *remainder sampling methods.* The integral part of the expected value is used to deterministically select strings. The fractional part can then be dealt with using other methods. For example roulette wheel selection can be used to produce a method with zero bias and lower bounded spread. To gain minimum spread the fractional expected value can be set to zero after each spin. Unfortunately this method produces bias in favour of smaller fractions. Another problem with these methods that still use the roulette wheel selection for part of the algorithm is that they will have $O(n \log n)$ time complexity [5].

The *stochastic universal sampling* method is suggested by Baker as a method which has zero bias, minimum spread and $O(n)$ time complexity. It is a system similar to roulette wheel selection. The difference is that a spinning wheel with n equally

[4]The term *replacement* here means that once a string has been selected it is put back into the population so it can be selected again.

spaced pointers is used. In a single spin all n parents are selected [5].

A good sampling algorithm like stochastic universal sampling should benefit all GAs because reducing bias and spread matches implementations more closely to the theory. In some cases bias is useful to speed up convergence near the end of a run or to prevent super individuals dominating at the beginning (see Section 2.7.7), but it should never be unknowingly introduced by a bad sampling algorithm.

2.7.3 Steady State Genetic Algorithms

The traditional GA typically replaces the whole population with new offspring each generation. This can have the undesirable effect that depending on the sampling algorithm used, the very best parents may not even reproduce. If a good sampling algorithm is used, such as stochastic universal sampling, then this will not happen. However, even if they do reproduce, there is a chance of crossover and mutation destroying these strings [21]. One solution that has been used to solve the problem partially is the automatic insertion of the best string into the next generation. This is called *elitism* [46]. Another method, which will be described here, is to overlap populations. This is known as a *steady state* genetic algorithm.

A parameter known as the *generation gap* (G) has been introduced, where Gn is the number of strings replaced each generation in a population of size n. A value of $G = 1$ is the standard GA where the whole population is changed each generation. A GA with $G < 1$ is called a steady state GA. Typically one or two strings are replaced per generation (i.e. $G = 1/n$ or $G = 2/n$ respectively). As with $G = 1$, parents are selected based on their fitness. The resulting offspring are added back into the population but they do not replace their parents. Rather some strategy is used to remove other strings. A typical method is to remove strings from the population which have the lowest fitness. With the steady state method it is also possible not to keep duplicates. This ensures as diverse a population as possible within the constraints of the population size.

The steady state methods have been shown in practice to perform well on some classes of problems [21] and there has been some theoretical analysis of the changes

in GA performance introduced by steady state GAs [25]. It is clear that when steady state replacement is combined with different worst string deletion methods and scaling methods, a very different result can be achieved compared with that of the standard GA.

2.7.4 Different Crossover Methods

A number of different crossover methods have been proposed. Often the problem domain determines what works best. Some specialized operators have been developed for problems like the TSP which will be studied in Chapter 3. In this section more general crossover operators are considered. As a motivation for looking at other operators, consider the case where we have two fit schemata in a chromosome, each at opposite ends, and this chromosome evaluates as fit only when both schemata are found together. The normal crossover operation can never combine these with another chromosome that has a good schema in the middle, say.

2.7.4.1 Two Point Crossover

One way to alleviate this problem is to use a 2-point crossover. In this method two points are selected in the strings and alleles are swapped between these two points [81].

2.7.4.2 Uniform Crossover

Uniform crossover is an even more extreme version. Here a crossover mask is randomly generated for each crossover operation. Each bit in the mask determines if a bit should be swapped between parents to produce the new offspring [81]. The problem with this approach is that it would seem that in terms of schema analysis there is absolutely no motivation for this operator as there will be on average $l/2$ crossover points in a string of length l, giving any schema of order two or more very little chance of survival. However, there have been indications that having more than two crossover points can be beneficial [28, 84]. One benefit that uniform crossover offers is that it is not biased in its disruption of schemata — all schemata of order k are disrupted with equal

probability no matter what their defining length. To reduce the disruptive effects, one solution is *parameterized uniform crossover* where the probability of swapping bits is adjustable. This enables precise control of the disruptive effects of crossover [81].

2.7.5 Mutation and Crossover Probability

There has been much research into the different rates at which mutation and crossover should be applied. There exist values that work rather well for a wide range of problems that have been determined experimentally. Research has shown that GAs are stable over a large variation in mutation and crossover probability. However, these generic values can clearly never be optimal for all problems and it is always possible that particular combinations of crossover operators, search space and other factors could require different values.

One approach is to run a GA to determine these values. This meta-GA has as its objective function the GA for which we are trying to determine the rates for mutation, crossover or any other generic operator [42]. Encoded into the string will be the values for mutation and crossover. This approach can be very computationally expensive as we have to perform a whole GA run for each objective function evaluation. In general this is only useful if the results can be reused.

Another approach [20] is to adjust the rate of application of the operator on its success. The quality of its output then determines how often it is applied to future strings. This technique can be applied to any number of operators making it possible to evaluate the performance of a number of operators at once. Factors that can be used to evaluate operator quality include comparative fitness of the strings generated and diversity of offspring produced.

2.7.6 The Inversion Operator

The inversion operator operates on a single string. Two random points are selected and the string between these two points is reversed. This operator is actually biologically motivated and is part of the original GA proposed by Holland but has not found much application in general [21]. For the inversion operator to be useful it must be applied

to encodings in which the meaning of an allele is not determined by its position in the string. This can be achieved by tagging alleles in the string so that when a string goes through inversion the alleles retain the same meaning. In an encoding like this the inversion operator does not change the value of a string, but since the genes move around within the string it changes the way in which crossover affects the string. Inversion can reorder the genes so that better building blocks can be constructed. It does, however, add complications in that using simple crossover can result in strings with duplicate or missing genes. Some extra logic has to be used to cope with this problem [33].

Inversion is also useful in problems where the relative positions of alleles, rather than their absolute positions, are relevant. This is the case with problems like the TSP and will be discussed in Chapter 3.

2.7.7 Scaling and Ranking

When maximizing a function using a GA great performance changes can be brought about by seemingly minor changes to the function [21, p31]. If for example a function $f(x) = x^2$ optimized over the interval $[0, 2]$ were changed to $f'(x) = x^2 + 10000$ the optimum remains the same but if this modified function is used as the fitness function in a GA there will be a marked difference in performance. This is because the relative difference between the points is no longer very big so there will be a much smaller difference in the sampling rates between weak and strong parents. The net result is slow convergence.

2.7.7.1 Scaling

To solve this sort of problem a number of different approaches can be used. A simple solution which works in the above case is to subtract the minimum overall value from all fitness functions before selection.

Often it is also desirable to adjust the calculated difference between strings during the GA run. A common problem with GAs is that 'super' individuals at the start of a GA run can dominate the population. To prevent this it is possible to scale

fitness values [21, p32]. A similar procedure can be used at the end of a GA run to increase competition between a number of strings with similar fitness values.

2.7.7.2 Ranking

Ranking does away with raw fitness functions. Rather, fitness is used to rank the strings. A function is then applied to the ranked value to get a ranked fitness value. The function used can be linear, quadratic or some other monotonic function. In *Linear Normalization* the string with the highest raw fitness value is assigned a constant value K. The next best string is assigned $K - D$, where D is some constant value. This process continues so that the string ranked n-th is assigned a value of $K - D(n - 1)$ [21, p31].

2.7.8 Hybrid Genetic Algorithms and Domain Specific Knowledge

GAs can be combined with standard optimization and search techniques to achieve better results [12]. For example, if a good local optimization algorithm exists for a domain, it may be worth using this as a replacement for, or in addition to, the mutation operator. This type of change is naturally very domain-dependent. Care must be taken that the GA is still playing its proper role. There is no point in creating an algorithm which will be out-performed by hill climbing with random restarts.

Specific modifications can be made to a GA to improve its performance in a particular problem domain. This is done by taking into account domain-specific knowledge that a GA cannot simply pick up from the fitness function. This extra knowledge can be used in the initial population generation and in the design of crossover and mutation operators. We will see this in Chapter 4 when we look at crossover operators for the TSP.

The only method of population initialization discussed so far is that of random generation of individuals. If done correctly, random initialization should give a good distribution of allele values, and so provide a basis from which to explore the search space [71]. There are, however, cases where the population can be successfully seeded

using another (preferably) fast algorithm so that the GA performs more quickly. It is also useful in the case where a GA is to be used to improve on an existing solution. An example of this is in engineering design where GAs have been used to help search for better designs. Using a population already seeded with existing good designs enables the GA to search better designs that are nearby in the search space [71].

2.8 Summary

Genetic Algorithms contain aspects not found in other search and optimization methods. Their ability to work on a population of points at the same time may, with careful coding of the parameters into a string, provide building blocks that give the GA an idea of the "shape" of the search space. The idea of *implicit parallelism* and the probabilistic nature of GAs ensure that they are robust and deal with convoluted search spaces and perform well. The theory behind GAs provides some understanding of how they perform and what can be done with them. A number of modifications can be made to the basic GA and it can be hybridized with more conventional techniques. Often such changes can only be justified by the success of the resulting algorithm.

The simplicity of the GA is one of its main assets. This makes it possible to adapt it to many different environments. For example it is easy to use GAs on parallel hardware or in distributed environments.

Chapter 3

The Travelling Salesperson Problem

3.1 Introduction

The Travelling Salesperson Problem (TSP) can be simply stated as the problem of finding the shortest route for a salesperson starting at a home city to visit a list of cities and returning home without entering any city twice. A mathematical statement of the TSP is given later in Definition 1. This chapter includes a look at why the TSP is a hard problem using some computational complexity theory. This will provide motivation for the application of heuristic techniques to the solution of the TSP. A number of these techniques will be described in the rest of the chapter.

3.1.1 History and Background

Some history of the TSP is essential to understand the importance of this problem in computational complexity and operations research. This section is based upon [2, 49] which contains a more complete chronology of the developments around the TSP.

It seems that the term "travelling salesman problem" was first mentioned in a mathematical context between 1920 and 1932 but it is not clear who first brought this term into the mathematical literature. What is known is that Merrill Flood publicized the term in the 1940s. At the time the new subject of linear programming

was generating combinatorial optimization problems such as the assignment problem. The TSP was interesting because it was similar but seemingly harder to solve. The TSP has also always attracted attention because of the easy statement of the problem and simply because of the name.

While problems like the assignment problem have a simple statement and solution using linear programming, problems like the TSP require a large number of inequalities and also require that the variables take on only integer values, making the TSP an integer programming problem. The paper *Solutions of a large-scale traveling-salesman problem* by Dantzig, Fulkerson and Johnson was published in the *Journal of the Operations Research Society of America* in 1954 [2]. A solution to a 49-city problem, found using string on a model, was proved optimal using techniques that would later be generalised to the branch and bound procedure. Branch and bound has since proved useful in the solution of other combinatorial optimization problems that arise from integer programming.

Despite developments in achieving a solution to the TSP, it became clear by the end of the 1960s that the TSP and other hard combinatorial optimization problems were more difficult than problems like the assignment problem which could be solved by an algorithm in polynomial time. This lead to papers of Cook, Karp and Levin [49] in the early 70s that showed the equivalence of hard problems and the definition of $\mathcal{NP}$-hard (see Section 3.3).

3.1.2 Significance

The TSP is significant for two reasons. Firstly, it is a typical combinatorial optimization problem which means that its study is of great theoretical interest. The history of the TSP shows how important the TSP was in the development of the theory and solutions in combinatorial optimization. Secondly, there are a number of real world applications which are related to the TSP. These can be stated as the TSP or variations of it. Examples are computer wiring, vehicle routing and job sequencing. These applications and some others will be described in Section 3.4. Any improvement in techniques to solve the TSP help the speed and accuracy of solutions for these and

many other applications.

3.2 Graph Theory Representation of the Travelling Salesperson Problem

The TSP is most often analysed in a graph theoretic context and is closely related to a number of other problems in graph theory. In this section, some graph theory definitions will be given, enabling the discussion of the TSP in the rest of the chapter. For this [14] and [75] and were used.

A graph G is a finite nonempty set of objects called *vertices* or *nodes* denoted by $V(G)$ together with a set of objects called *edges* denoted by $E(G)$. An edge e is a pair of distinct vertices denoted by $e = u, v$ or $e = uv$. Edge e is said to *join* the vertices u and v. The vertex u is *adjacent* to v while u (or v) is *incident* to e. The *degree* of a vertex v (deg v) is the number edges incident with v. A graph G is said to be complete if every vertex is adjacent to every other vertex. A complete graph on n vertices is denoted by K_n. A graph H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. It is a *spanning subgraph* if $|V(H)| = |V(G)|$.

A *directed graph* or *digraph* D is a finite nonempty set of objects called *vertices* denoted by $V(D)$ together with a set of ordered pairs called *arcs* or *directed edges* denoted by $E(D)$. A digraph D is said to be complete if, for every two distinct vertices u, v in $V(D)$, both (u, v) and (v, u) are elements of $E(D)$. A complete digraph on n vertices is denoted by D_n.

A set of edges $W = \{v_1v_2, v_2v_3, \ldots, v_{k-1}v_k\}$ is called a *walk* or a $[v_1, v_k]$-walk. If a walk does not contain any repeating vertices it is called a *path*. If $v_1 = v_k$ for some $[v_1, v_k]$-walk then it is called a *closed walk*. A set of edges $C = \{v_1v_2, v_2v_3, \ldots, v_{k-1}v_k, v_kv_1\}$ with $v_i \neq v_j \; \forall i \neq j$ is called a *cycle*[1]. A vertex u is *connected* to v in G if there exists a $u - v$-walk in G. A graph G is *connected* if every two vertices in G are connected. A connected graph containing no cycles is called a *tree*.

[1] *Diwalks*, *dipaths* and *dicycles* are defined in a similar way for digraphs with the additional requirement that arcs are directed in the same direction.

A graph (digraph) G is said to be *Hamiltonian* if it has a cycle (dicycle) containing all the edges of G. This cycle (dicycle) is called a *Hamiltonian cycle (dicycle)* or *Hamiltonian tour* or simply a *tour.*

A *weight function* $c : E(G) \rightarrow Q$ can be associated with a graph (digraph) G. For each edge (arc) $uv \in E(G)$ the weight function, c_{uv} (or $c(uv)$), defines the weight of edge (arc) uv. The weight of a set of edges (arcs) $S \subseteq E(G)$ is defined as $c(S) = \sum_{uv \in E(G)} c(uv)$. The weight of a tour is called its *length* and a tour with the smallest length is called the *shortest tour.*

The TSP can now be stated mathematically in graph theoretic terms. Definition 1 is the definition for the TSP. In the case of the TSP the weight of each edge can be interpreted as the distance between cities.

Definition 1 (Asymmetric Travelling Salesperson Problem) *Given a complete weighted digraph D_n with $n \geq 3$ and arc weights c_{uv} find the shortest Hamiltonian tour in D_n [75, p6].*

It is possible to restrict the distances between cities to integers without any loss of generality. Reals are approximated as irrationals in computers and irrationals can always be represented by integers if multiplied by a large enough scaling factor.

3.2.1 Specializations of the Travelling Salesperson Problem

A number of different specializations and generalizations of the TSP exist. The TSP defined in Definition 1 is referred to as the asymmetric TSP because the distance from city u to city v need not be the same as the distance from city v to city u. Three commonly encountered versions of the TSP are given below [52]:

Symmetric TSP The distances between cities are the same in both directions. That is, $c_{ij} = c_{ji}$ for all i, j. This is what most people think of as the TSP.

Symmetric triangle inequality TSP This is the Symmetric TSP with the added restriction that all distances obey the triangle inequality. That is, $c_{ij} + c_{jk} \geq c_{ik}$ for all i, j, k.

Euclidean TSP Cities are given as points with integer coordinates in a two dimensional plane. Distances are calculated using the Euclidean metric:

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

It should be noted that the euclidean TSP is a symmetric triangle inequality TSP and the symmetric triangle inequality TSP is a symmetric TSP. Many more variations of the TSP exist. One can refer to Johnson and Papadimitriou for a list of other special cases [52, p59]. The rest of this research will deal with the symmetric TSP and any further unqualified mention of the TSP refers to the symmetric TSP. The reason for this restriction to the symmetric TSP is to reduce the scope of the project to a reasonable size and to make comparison with other techniques easier. Just as there are restrictions to the TSP, there are also extensions so a decision has to be made where to restrict the study. The asymmetric TSP is not studied as much as the symmetric TSP. Some good heuristic solutions like the Lin-Kernighan heuristic apply only to the symmetric case [60]. Restricting this dissertation to the symmetric TSP contains the scope of the work and allows existing techniques for the symmetric TSP to be incorporated into the work.

3.3 Complexity Theory and the Travelling Salesperson Problem

How difficult is the TSP and how well does a particular algorithm perform on the TSP? A study of the theory of $\mathcal{NP}$-completeness provides some understanding in both these regards. Interestingly, the TSP was instrumental in the development of the theory [52]. This section does not look at the space complexity of the TSP. This will be discussed in Section 3.10.3 when data structures for the TSP are examined.

3.3.1 Worst Case and Average Case Performance

When considering how good an algorithm is, different measures of its time performance can be used. Either the worst case performance can be considered or the average case

performance. Worst case performance guarantees can be misleading as there may be a relatively small number of pathological cases. An example commonly mentioned is that of the simplex method by Dantzig for solving linear programming problems. This method can be used to efficiently solve large linear programming problems but requires exponential running time in some cases [75]. It does however give an idea of the worst that can be expected and provides a good theoretical basis to analyse both problems and algorithms. The elegant theory of $\mathcal{NP}$-completeness is the result of examining worst case performance.

Average case performance is also very interesting but can be difficult to calculate. This is because most efficient algorithms for complex problems like the TSP have different interdependent stages which are difficult to analyse. This means that for some algorithms the average case performance is difficult or impossible to calculate [52, p43]. Karp and Steele deal with the average case performance of some algorithms and some cases of the TSP [56].

3.3.2 Processor Independence

When comparing the performance of different algorithms it is useful to be able to remove any dependence on processor speed. The O-notation in Definition 2 is commonly used to compare performance. The O-notation removes any dependence on the speed of the computer used to perform the procedure.

Definition 2 (O-notation) *[75] Let $f : N \to N$ and $g : N \to N$ be given. We say that f is $O(g)$ if there exist positive constants c and n_o such that*

$$0 \leq f(n) \leq c.g(n) \text{ for all } n \geq n_o$$

3.3.3 Definition of a Good Algorithm

The time complexity of an algorithm A is given by $t_A(n)$, where $t_A : N \to N$. An algorithm is said to have polynomial time complexity if there exists a polynomial p such that $t_A(n) = O(p(n))$. Otherwise it is said to have exponential time complexity [75].

An algorithm is called a *good* algorithm if it has polynomial time complexity. In practice, polynomial time algorithms may still be too inefficient for real world problems. There are some functions which are neither exponential nor polynomial, such as $n^{\log n}$, but since these are not polynomial they will be included with exponential time complexity algorithms [52].

Polynomial time has a number of desirable properties. There are several different computing models, such as the Turing machine, which are capable of simulating each other with only a polynomial change in speed. This means that if an algorithm has polynomial time complexity on one machine, it will have polynomial time complexity on all machines. Polynomials also behave well mathematically: if polynomials are added, multiplied or composed they remain polynomials [52]. This leads to the following useful definition.

Definition 3 (polynomial reducible) *[75] Let A be an algorithm for the solution of a problem $\mathcal{B}$. We say that a problem $\mathcal{C}$ is* polynomial reducible *to problem $\mathcal{B}$ if $\mathcal{C}$ can be solved in polynomial time by an algorithm that uses A as a subroutine where each subroutine call of A counts as only one step.*

3.3.4 Decision Problems

A decision problem is a problem that requires only a yes or no answer. The basic theory for complexity is developed for decision problems. Problems like the TSP can be stated as a decision problem to fit into this framework. Definition 4 gives a decision version of the TSP. Later it will be explained how the theory can be extended to combinatorial optimization problems like the TSP.

Definition 4 (TSP decision) *[75] Given K_n with edge weight c_{uv} and a number b decide if there exists a Hamiltonian tour in K_n with length less than or equal to b.*

Clearly an algorithm that solves the TSP will also solve the TSP decision problem in the same time. Thus there is a trivial polynomial reduction from TSP to TSP decision. Interestingly the converse is also true: there exists a polynomial reduction from TSP decision to TSP. The following explanation is based upon [52, 75]. Algo-

rithm 2 finds the actual length of the optimal tour. It makes use of the assumption that the distances used are integral[2]. It uses TSP decision. Since it uses a binary search it makes $O(\log n)$ calls to TSP decision. Given the length of the optimal tour, Algorithm 3 finds an optimal tour in the graph. It does this by setting each edge weight in turn to $\overline{c}.n$, where $\overline{c}$ is the largest weight of an edge in the graph, and checking if this affects the tour length using TSP decision.

Algorithm 2 (TSP Length)

Set $L = -\overline{c}.n$ and $U = \overline{c}.n$, where $\overline{c}$ is the largest weight of an edge in the graph.

WHILE $L < U$

Set $b = \lceil \frac{L+U}{2} \rceil$.

If there exists a Hamiltonian tour of length at most b then set $U = b$, otherwise set $L = b + 1$.

END WHILE

■

Algorithm 3 (TSP Tour)

Let U be the optimal tour length found by algorithm *TSP Length.*

FOR all $u = 1, 2, \ldots, n$ and all $v = 1, 2, \ldots, n$ perform the following steps.

Set $s_{uv} = c_{uv}$ and $c_{uv} = \overline{c}.n + 1$.

If there does not exist a Hamiltonian tour of length U in the modified graph then restore $c_{uv} = s_{uv}$.

END FOR

The edges of the graph which have not been altered give the edges of an optimal tour.

■

[2]As discussed in Section 3.2 this does not pose a problem.

3.3.5 $\mathcal{P}$ and $\mathcal{NP}$ Problems

$\mathcal{P}$ is defined to be the class of decision problems that have algorithms with polynomial time complexity. Another class of decision problems is defined using the property of non-determinism. Non-determinism is modeled on a theoretical construct — the non-deterministic Turing machine. This can be thought of as a computer that has an extra instruction that can split execution into two branches without any loss of performance in each branch [52]. As a result it is possible to produce an exponential number of branches in polynomial time. The class $\mathcal{NP}$ is defined to be those decision problems that can be solved in non-deterministic polynomial time if the answer is yes[3].

Given a TSP decision problem with a bound b on n cities there are no more than $n!$ possible tours. By simple addition, it is possible to check in only linear time whether a given tour meets the bound b. It is possible to use the power of non-determinism to split the problem into $n!$ problems in only $\log_2 n! = \log_2 1 + \log_2 2 + \cdots + \log_2 n \leq n \log_2 n$ steps which means that this process also takes polynomial time. The sum of two polynomial time procedures is still of polynomial time. Thus the TSP decision problem is in $\mathcal{NP}$.

Clearly $\mathcal{P} \subseteq \mathcal{NP}$ but is $\mathcal{P} = \mathcal{NP}$? This is a long-standing problem in computation complexity. It seems very unlikely that $\mathcal{P} = \mathcal{NP}$, particularly when one considers how much more difficult problems like the TSP are than problems which are in $\mathcal{P}$. Researchers have been searching for good algorithms for the TSP for a long time without success, but it cannot be shown that one does not exist unless it is proved that $\mathcal{P} \neq \mathcal{NP}$ [52]. This conjecture looks like it may never be proven but it has been shown that $\mathcal{P} \neq \mathcal{NP} \Leftrightarrow$ TSP decision $\notin \mathcal{P}$. This result will be examined in the next section.

3.3.6 $\mathcal{NP}$-completeness

$\mathcal{NP}$-complete is the class of decision problems that are in $\mathcal{NP}$ and are in some way equivalent to each other. It is defined in Definition 5.

[3]The amount of time required to return a no answer is not of concern in the definition of $\mathcal{NP}$.

Definition 5 ($\mathcal{NP}$-complete) *[52] A problem A is $\mathcal{NP}$-complete if it is a member of $\mathcal{NP}$ and every problem in $\mathcal{NP}$ has a polynomial reduction to A.*

TSP decision is a member of $\mathcal{NP}$-complete. For a proof of this consider [52].

3.3.7 $\mathcal{NP}$-hard Problems

The results so far have dealt only with decision problems. By definition the TSP is not $\mathcal{NP}$-complete because it is not a decision problem. The TSP is actually a member of the class of $\mathcal{NP}$-hard problems which is the class of optimization problem to which all $\mathcal{NP}$ problems are reducible [52]. This is true because TSP decision can be reduced to TSP and TSP decision is a member of $\mathcal{NP}$-complete. An optimization problem having a polynomial reduction to a problem in $\mathcal{NP}$ is called $\mathcal{NP}$-easy. Since TSP can be reduced to TSP decision, which is in $\mathcal{NP}$-complete, TSP is also $\mathcal{NP}$-easy. A problem that is both $\mathcal{NP}$-hard and $\mathcal{NP}$-easy, like the TSP, is called $\mathcal{NP}$-equivalent [75]. We then have these results:

$$\mathcal{P} \neq \mathcal{NP} \Rightarrow \text{no } \mathcal{NP}\text{-hard problems can be solved in polynomial time} \quad (3.1)$$

$$\mathcal{P} = \mathcal{NP} \Rightarrow \mathcal{NP}\text{-easy} \subseteq \mathcal{P} \quad (3.2)$$

Thus the final result of this reasoning is that from Implication 3.1, in worst case analysis, no polynomial time algorithm exists for the TSP, if it is assumed that $\mathcal{P} \neq \mathcal{NP}$. There is also a strong indication that there will never be a good algorithm for the TSP because if one is found, then $\mathcal{P} = \mathcal{NP}$.

Even though the TSP is $\mathcal{NP}$-hard, special cases need not be. Some special cases of the TSP were mentioned in Section 3.2.1; unfortunately all are $\mathcal{NP}$-hard. Many more variations of the TSP exist but the interesting cases are all $\mathcal{NP}$-hard. For a list of other special cases, and proofs that they are $\mathcal{NP}$-hard, refer to [52].

3.4 Practical Applications of the Travelling Salesperson Problem

The TSP as stated appears to apply to a useful real world application, but in reality practical applications of the TSP are different. Some require transformation in order to be formulated as TSPs. Some contain the TSP as a component. This section discusses a few of the practical applications to which the TSP can be applied.

3.4.1 Computer Wiring

A number of modules with pins are placed on a circuit board. A subset of n pins needs to be connected by wires with the restriction that no more than two wires can be connected to each pin (perhaps to make manufacturing easier). The pins must be connected using a minimum amount of wire to reduce cost and interference. Let c_{ij} denote the distance between pin i and pin j. Without the restriction on the number of connections it would be possible to use a good minimum spanning tree algorithm. Instead it is necessary to find a minimum Hamiltonian path. To transform this to a $(n+1)$-city TSP it is only necessary to add dummy pin 0 such that $c_{i0} = c_{0i} = 0$ for all i [32].

3.4.2 Drilling of Printed Circuit Boards

Drilling holes in a printed circuit board (PCB) is one of the steps required in its production. Moving the drill head from one position to the next takes time so it makes sense to drill holes in an order that minimizes the total distance moved by the drill head. Holes of different diameters often have to be drilled into the board. Assuming that the drill bit cannot be changed without returning the drill head to an origin, the drilling of holes can be solved as m separate TSPs if there are m different hole sizes [75, ch11].

3.4.3 Job Sequencing

A set of n jobs needs to be sequenced on a machine. In order to perform job j the machine must be in state S_j. The state of the machine is any physical characteristic of the machine such as position, temperature or paint colour. The time to transform the machine from state S_i to state S_j is given by c_{ij} and the time to perform job k is given by p_k. Thus the time to perform job j following job i is $t_{ij} = c_{ij} + p_j$. The quickest job sequence must be found, given that the machine must start and end in state S_0. So given a sequence of jobs as a cyclic permutation π of $0, 1, \ldots, n$ then the total time required to do all n jobs given that $p_0 = 0$ is

$$\sum_{i=0}^{n}(c_{i\pi(i)} + p_{\pi(i)}) = \sum_{i=0}^{n} c_{i\pi(i)} + \sum_{i=0}^{n} p_{\pi(i)}$$

Since $\sum_{i=0}^{n} p_{\pi(i)}$ is constant for all variations of π it follows that this problem can be solved as an asymmetric TSP, or as a symmetric TSP if state transformations take equal time in reverse [32].

3.4.4 The Order-Picking Problem in Warehouses

Given a warehouse and a subset of n items to be picked for an order, we want to find the sequence in which items should be collected to minimize the total time required. To solve this as a TSP the storage bins become vertices on a graph with distances between the vertices equal to the time required to move between bins. Given a starting point for the collection this problem can be formulated as a $(n + 1)$-city TSP [75, p36].

3.4.5 Vehicle Routing

Given n customers and m vehicles how is q_i delivered to customer i? Each vehicle k has a capacity of Q_k. For each vehicle a single route must be found, starting and ending at one central depot, so that each customer gets their goods, the capacity of any vehicle is not exceeded and total travel costs are minimized. The vehicle routing problem is more complicated than the TSP but it is possible to design a solution that has the TSP as a subproblem [16].

3.5 Travelling Salesperson Problem Size

With improvements in techniques and computing power it is now possible to find optimal solutions (and know they are optimal) to some large problems. For example, problems as large as 4461 nodes have been reported solved to optimality [75]. Unfortunately existing optimal problem solvers require very large amounts of CPU time and they are not stable — the solution of one problem of more than n nodes does not mean all problems under size n can be solved as easily [55].

Heuristic solutions to the TSP have also improved as different techniques have become available. When the Lin-Kernighan method was proposed the results obtained then suggested that the run time grows at a rate of $n^{2.2}$ [60]. The Lin-Kernighan and similar methods can handle problems with around 6000 cities and produce results close to optimal results (2% above best known lowest bound) in reasonable time (under an hour on a workstation) [75]. The trade-off with these methods is normally time versus solution accuracy.

3.6 Heuristic Solutions of the Travelling Salesperson Problem

As the TSP is in the class of $\mathcal{NP}$-hard problems, two different approaches to its solution can be taken. One approach is to attempt to find the best algorithm to solve the problem exactly, knowing that it will take a long time, at least in some cases. Being $\mathcal{NP}$-hard does not mean that every run of the algorithm will require exponential time but it does mean that it will be required in some cases. The average case performance of the TSP is still an open question. Some believe it to be polynomial while others think it will be exponential [6]. Existing algorithms for the TSP that find exact solutions are still very expensive and not practical for general use on large problems [55]. An alternative is to solve the TSP using a heuristic that will find approximate solutions in reasonable time. A number of algorithms have been developed, some of which will often produce answers within a few percentage points of the optimum solution. Some will be discussed in the sections that follow. An interesting result, to

put the performance of these heuristics into perspective, is that of Theorem 1.

Theorem 1 (Sahni & Gonzalez, 1976 [53])
Suppose there exists a polynomial-time heuristic A for the TSP and a constant r, $1 \leq r < \infty$, such that for all instances I of the TSP,

$$\mathrm{A}(I) \leq r\,\mathrm{OPT}(I).$$

Where $\mathrm{OPT}(I)$ is the length of an optimal tour. Then $\mathcal{P} = \mathcal{NP}$.

Theorem 1 shows that, assuming $\mathcal{NP} \neq \mathcal{P}$, either a heuristic will require exponential running time sometimes or if polynomial-time performance is required no guarantees can be placed on the accuracy of the results. It does not, however, say anything about average case performance or accuracy so the outlook for a particular heuristic may be much better on average. Later it will also be shown that if only triangle inequality TSPs are considered then some bounds can be obtained on how bad the solution is.

Since the GA can be used only as a heuristic for TSP search we cannot expect exact solutions. For this reason we will concentrate on the traditional heuristic algorithms for the TSP. These are the algorithms against which the GA will compete. The analytical treatment of the performance of heuristic TSP algorithms is complex and in some cases has not been possible [53]. Some comments will be made about time complexity and performance guarantees. In the case of a GA-based TSP solver it is definitely very complex to carry out an analytical or even probabilistic analysis because of the many interdependencies in the GA. For this reason comparison of heuristics will proceed empirically.

The next two sections that follow look at a number of techniques that have been tailored to the solution of the TSP. This does not cover general-purpose search heuristics that have been used on the TSP such as simulated annealing [11] and ant systems [26].

Heuristics for the TSP can be divided into two classes. *Tour construction procedures* construct a feasible tour from scratch while *Tour improvement procedures* attempt to improve a given feasible tour. The next two sections will look at a number of these techniques in some detail.

3.7 Tour Construction Procedures

Tour construction procedures in this section build tours using some heuristic. Once the tour is built, no attempt is made to improve the quality — this will be looked at in Section 3.8. Often the running time for these procedures can be analysed analytically because vertices are added one by one to form a new tour making time complexity calculations possible. Ideally it should be possible to provide guarantees for the solutions produced by a heuristic. This is possible with some of the heuristics described in this section.

3.7.1 Nearest Neighbour

One obvious heuristic for constructing a tour is, at each step, to move to the nearest city not yet visited. This is an easy-to-implement algorithm. The running time is $O(n^2)$. This heuristic produces relatively good starts to the tour but fails near the end when large edges need to be added to get to the remaining cities [53]. In Figure 3.1 is a tour generated by the nearest neighbour heuristic, in which this problem can be clearly seen. One theoretical result which should limit our expectations for the nearest neighbour algorithm is given in Theorem 2. This theorem proves that no upper bound can be placed on the inaccuracy of the nearest neighbour tour for all TSP instances, even in the case of the triangle inequality TSP.

Theorem 2 (Rosenkratz, Stearns and Lewis [53])
For every $r > 1$ and arbitrarily large n, there exists an n-city triangle inequality TSP instance I such that

$$\mathrm{NN}(I) \geq r\,\mathrm{OPT}(I).$$

where $\mathrm{NN}(I)$ is the nearest neighbour tour length of I and $\mathrm{OPT}(I)$ is the optimal tour length of I.

A number of improvements can be made to the nearest neighbour algorithm, particularly to improve the running time. Some of these will be described in the following sections.

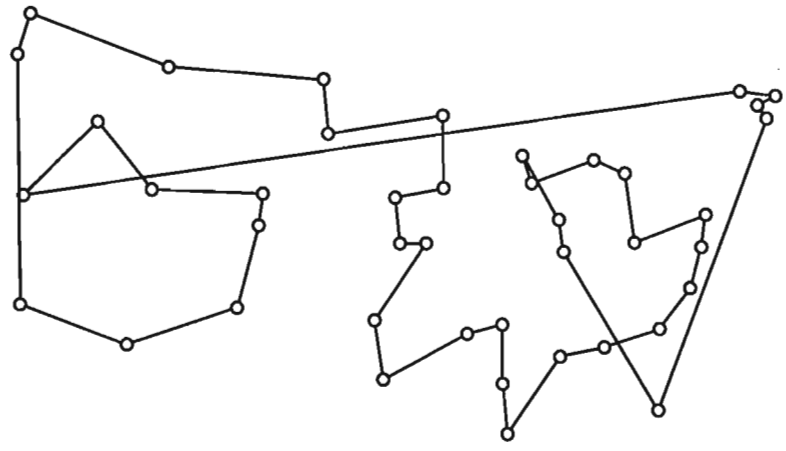


Figure 3.1: Nearest neighbour 42-city tour

3.7.1.1 Exploiting Subgraphs

The basic nearest neighbour heuristic runs slowly because of the large number of edges that need to be considered each time. Most of these edges will be much too long and need not be considered. This can be achieved by constructing a spanning subgraph, called a *candidate set*, of the complete graph by removing some edges between distant vertices. This technique can be successfully used with other heuristics too [75]. Examples of subgraphs are:

Nearest Neighbour Subgraph The *k nearest neighbour subgraph* is constructed by removing all edges from a city except those connecting it to its k nearest neighbours. The only problem with this technique is that it may disconnect the graph so a check needs to be made that the graph remain connected.

Delaunay Candidate Set The *delaunay candidate set* is another set. Its construction is complicated and requires the introduction of a number of geometric concepts. For details see [75].

3.7.1.2 Precomputed Neighbours

When using a candidate subgraph it is possible to limit the search for the nearest neighbour to just those adjacent in the subgraph. If this fails then the nearest neighbour must be computed amongst all free nodes. This improves performance in the

average case time complexity but not in the worst [75].

3.7.1.3 Neighbours of Predecessors

As for precomputed neighbours, the candidate subgraph is first checked. If this fails then the neighbours of the predecessor are checked. If this again fails the next predecessor is checked. After backtracking to some limit the nearest neighbour must be computed amongst all free nodes because otherwise the current node will be rather distant from the inserted node. Again this method improves only average case time complexity [75].

3.7.1.4 Insertion of Forgotten Nodes

The nearest neighbour heuristic's primary failing is that it inserts very long edges at the end when the choice has been reduced. Insertion of forgotten nodes requires a candidate subgraph where the degree of each vertex is recorded. As a vertex is added to the partial tour the degree of adjacent vertices is decreased. If the degree of a vertex drops below some threshold (e.g., 2 or 3) the vertex is inserted immediately. The insertion point is determined by examining possible insertion points before or after neighbours in the candidate subgraph that have already been inserted into the tour [75].

3.7.2 Insertion Heuristics for Tour Construction

Insertion heuristics work by starting with a subtour which is then extended to a complete tour. The tour may begin as the trivial tour on a single vertex or may be provided with a substantial subtour by another heuristic. An insertion heuristic must decide what vertices are selected for insertion and where they are inserted into the subtour. In general, once selected the vertex is inserted into the tour at a position that will cause the smallest increase in tour length. Another technique is to insert the vertex as a neighbour of the nearest subtour vertex. This is generally called *addition* rather than insertion [11, p393]. The following subsections describe different strategies for selecting the node to be inserted.

3.7.2.1 Nearest Insertion

Select a vertex that is closest to the subtour vertices [11, p393].

3.7.2.2 Farthest Insertion

A number of options exist [11, p394], including:

1. Select a vertex whose minimal distance to a subtour vertex is maximal.
2. Select a vertex that is farthest from a subtour node.
3. Select a vertex whose maximal distance to a subtour vertex is minimal.

3.7.2.3 Cheapest Insertion

Select a vertex that, when inserted into the subtour, causes the smallest increase in subtour length. This is a computationally expensive method since the cheapest insertion point of each vertex has to be tracked and updated whenever a new vertex is inserted. This method can be sped up if only a partial update of cheapest insertion point information is performed on each insertion. This results in some loss in accuracy. For example if u has been inserted then for each non subtour vertex v reconsider only insertion points for v in the subtour neighbourhood of u [75, p83].

3.7.2.4 Random Insertion

Select a vertex at random for insertion [75, p83].

3.7.2.5 Largest Sum Insertion

Select the vertex whose sum of distances from the subtour vertices is maximal [75, p83].

3.7.2.6 Smallest Sum Insertion

Select the vertex whose sum of distances from the subtour vertices is minimal.

3.7.3 Candidate Subgraph Insertion Heuristics

As was the case with nearest neighbour heuristics it is possible to speed up insertion heuristics using a candidate subgraph. For most insertion heuristics the only change is to perform calculations on the candidate subgraph. If the subgraph does not provide enough connectivity to continue with the insertion of vertices then a random vertex is selected. A description of the operations required for nearest insertion follows, as well as descriptions of some of the other operations that require a slightly different approach [75, p64].

3.7.3.1 Nearest Insertion

Select a vertex that is connected to the subtour by the subgraph and is closest to the subtour vertices. If no such vertex exists then select a random vertex.

3.7.3.2 Cheapest Insertion

Select a vertex that is connected to the subtour by a subgraph edge that, when inserted into the subtour, causes the smallest increase in subtour length. If such an edge cannot be found then ignore the subgraph and perform the calculation again. Insertion information is updated only for vertices that are connected to the last inserted vertex.

3.7.3.3 Random Insertion

Select a vertex at random for insertion, giving priority to vertices connected to the subtour by the subgraph.

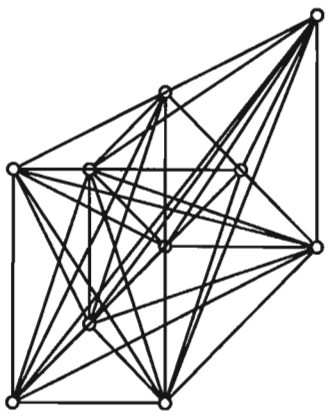
3.7.4 Spanning Tree Heuristics

A graph G is said to be connected if for every pair of vertices, it contains a path connecting them. A tree G is a connected graph containing no cycles. A spanning tree of a graph G is a subgraph of G that is a tree. In a weighted graph G a minimal spanning tree is a spanning tree of minimal length [53, 152].

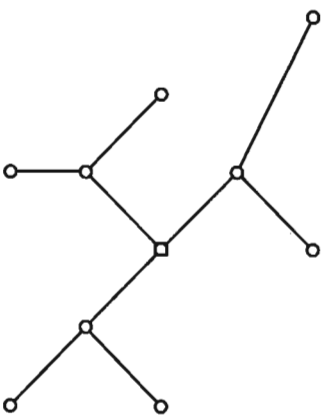
Given a tour of a graph G one can form a spanning tree by removing any one edge. Thus an optimal tour T_{opt} of G cannot be shorter than the length of a minimal

spanning tree. It is also possible to construct a tour from a minimal spanning tree. This process will be described below and each step is illustrated in Figure 3.2. The square vertex represents the home city.

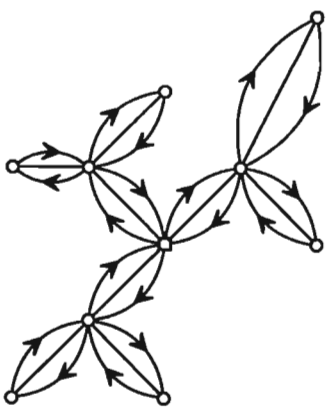
Start with a complete graph G (Figure 3.2(a)). Let T be a minimal spanning tree of G (Figure 3.2(b)). If a depth-first search is performed on T it will produce a walk W which visits each edge exactly twice (Figure 3.2(c)). The length of W will be twice that of the minimal spanning tree T. It is now possible to convert W to a tour T by replacing any section that backtracks over edges that have already been used by a single edge (Figure 3.2(d)). If the triangle inequality holds then a short cut edge to connect vertices u and v, say, can never be longer than the $[u, v]$-path it is replacing. Thus we have $|T| \leq |W| = 2|T|$. But $|T| \leq |T_{\text{opt}}|$ so $|T| \leq |T_{\text{opt}}|$ [53, p152].



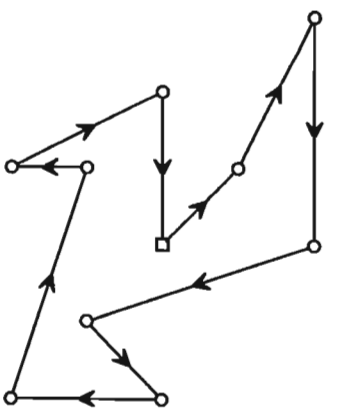
(a) Complete graph K_{10}



(b) Minimum spanning tree



(c) Depth-first search walk



(d) Hamiltonian tour of K_{10}

Figure 3.2: Spanning tree heuristic

The above method of constructing a tour using a spanning tree can be general-

ized by observing that the depth-first walk can be viewed as an Eulerian tour on the spanning tree where all the edges have been doubled. A graph contains an Eulerian tour if and only if all vertices have even degree [14]. Thus given a minimal spanning tree it is possible to produce an Eulerian tour if enough edges are added to give all vertices even degree. Once an Eulerian tour exists it is possible to turn it into a Hamiltonian tour as explained above [53].

The first method described ensures that each vertex has even degree by doubling the number of edges in the minimal spanning tree. Clearly it should be possible to give all vertices an even degree using a more carefully chosen set of vertices that doesn't increase the length of the graph as much.

Every graph has an even number of odd vertices because the sum of the degrees of all vertices in a graph is even[4]. It is thus possible to give all vertices even degree by adding a *perfect matching* in the vertices of odd degree. A perfect matching of a set of vertices W is a set of edges F such that each vertex in W is incident with exactly one edge in F [14]. A minimum weight perfect matching will obviously give the best results. Algorithm 4 makes use of the above technique to find a tour. The construction of the minimum weight perfect matching takes time $O(k^3)$ for a k vertex set and since there may be n odd-degree vertices this construction dominates the algorithm giving it $O(n^3)$ worst case time complexity [75].

Algorithm 4 (Christofides [53])
Let G be a complete weighted graph.

Construct a minimum spanning tree H in G.

Construct a minimum weight perfect matching on the set of odd-degree vertices of H and add this to H to get I.

Construct an Eulerian tour J in I.

Build a Hamiltonian tour K from J.

■

[4]Each edge is counted twice when calculating the sum of degrees of all vertices so the sum must be even.

For the triangle inequality TSP the following theorem holds

Theorem 3 (Christofides)

Given an instance of the TSP on which the triangle inequality is true Christofides's algorithm will produce tours that are no more than 1.5 times the length of an optimal tour.

3.7.5 Savings Method

The savings method is based on a method designed for vehicle routing problems [16]. As the TSP is a specialization of the vehicle routing problem it is possible to use the savings method to solve the TSP by considering it to be a vehicle routing problem with only one vehicle.

The savings method starts with a number of subtours of two vertices which are then connected to form larger and larger tours until a complete tour has been constructed. The tours that are merged are selected based on which merge will produce the biggest savings in tour length. Algorithm 5 contains an algorithm for the savings heuristic.

Algorithm 5 (Savings [75])

Select a vertex b called the base and construct $n-1$ tours $S = \{(b, u) : u \in V \setminus b\}$.

WHILE S contains more than one tour.

For each distinct $T_1, T_2 \in S$ calculate the savings that could be obtained by merging T_1 and T_2 by removing from each an edge adjacent to b, ub and vb say, and adding the edge uv.

Merge the two tours that have the largest savings, updating S by removing the original two tours and adding the new merged tour.

END WHILE

■

3.7.6 Comparison of Tour Construction Procedures

The average quality of the construction heuristics described here do not get better than 11% from optimality [75]. The best results are produced by the savings method which is 1–2% better than any other heuristic on average [75].

3.8 Tour Improvement Procedures

Tour improvement procedures require an initial complete tour. The tour can either be randomly constructed or can be produced using one of the tour construction procedures. Some improvement procedures like the Lin-Kernighan heuristic (see Section 3.8.5) work well even with random tours while some of the less effective procedures produce acceptable results only when paired with a good construction algorithm.

3.8.1 Node Insertion

The simplest tour improvement consists of removing a node and reinserting it in the most optimal position [75, p100]. This process can cycle through the nodes of the tour until no further improvement can be made. If there are n cities, then since each can be placed in $n-2$ other positions it takes time $O(n^2)$ to check all possibilities.

3.8.2 Edge Insertion

Edge insertion consists of removing an edge and reinserting it in the most optimal position in the tour [75, p100].

3.8.3 2-Opt Heuristic

The 2-opt move consists of removing two edges from a tour and reconnecting the tour so that the resulting tour is shorter [11, p399]. A tour is said to be *2-optimal* if all possible 2-opt moves have been applied. In the euclidean case a 2-optimal tour will contain no edges that cross, but 2-opt moves can also improve a tour with non-crossing

edges. Figure 3.3 illustrates a 2-opt move that shortens part of a tour. In this case the edges do not cross. To consider all 2-opt moves takes time $O(n^2)$.

The 2-opt heuristic was applied to the nearest neighbour tour in Figure 3.1 to give the 2-optimal tour shown in Figure 3.4. This is close to an optimal tour — only one node is out of place. This points to a possible improvement in the 2-opt procedure. The combination of 2-opt and node insertion can produce even better results [75] because most node insertion moves require two 2-opt moves and they may only produce an improvement in combination. This gives some motivation for looking at the r-opt improvement operator.

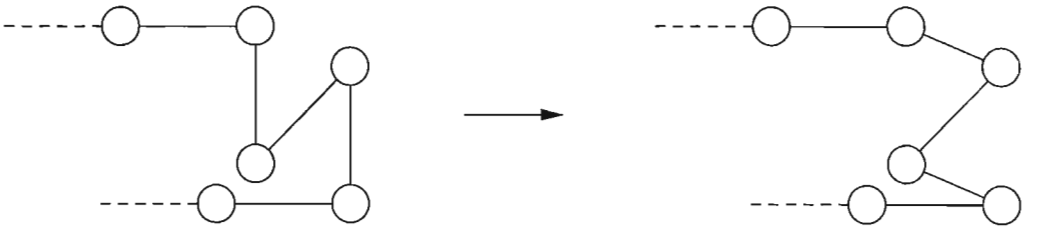


Figure 3.3: 2-Opt move

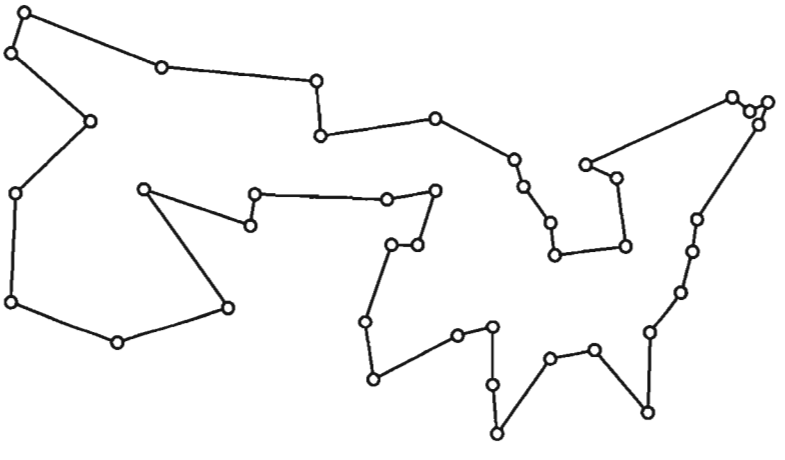


Figure 3.4: 2-optimal 42-city tour

3.8.4 r-Opt Heuristic

It is also possible to consider 3-opt moves where three edges are removed from a tour and it is reconnected in the most efficient fashion. This can then be generalized to r-opt moves where r edges are removed and reconnected to make a shorter tour. Once

all possible r-opt moves have been applied the tour is said to be *r-optimal.* To consider all r-opt moves requires $O(n^r)$ time which quickly makes this method impractical for large problems. Even 3-opt is very expensive so it is worth considering extending this method with optimizations which produce good results.

One improvement that can be made is to use candidate subgraphs as were used in some of the tour construction procedures. If only edges in the candidate subgraph are considered then a substantial drop can be obtained in the number of edges that need to be examined.

3.8.5 Lin and Kernighan Heuristic

A particularly well-known tour improvement procedure is that of Lin and Kernighan [60]. This makes use of an edge exchange procedure which is used as a modification on the r-opt moves. Using simply 2-opt moves it is possible to transform a tour to any desired tour. It may take a large number of moves but it can be done. Once a graph is 2-optimal it may be possible to reach a better tour using 2-opt moves but this requires that the tour become temporarily longer. The Lin-Kernighan heuristic deals with this case by searching tours that may be temporarily longer.

Lin and Kernighan's algorithm uses variable values for r at each iteration. This is a more powerful algorithm that produces near optimal solutions. The basic ideas in the Lin-Kernighan algorithm can be used in a number of different ways [61, 75].

3.8.5.1 Iterated Lin-Kernighan Heuristic

The results produced by the Lin-Kernighan heuristic can be improved by perturbing them and then re-running the Lin-Kernighan heuristic. This process can be repeated a fixed number of times or until no further improvement is obtained. This is known as the *iterated Lin-Kernighan heuristic.* Any tour improvement heuristic can be iterated in this way but high quality heuristics like the Lin-Kernighan algorithm give worthwhile results for this computationally expensive procedure [75, p129]. A perturbation method suggested by Reinelt is to perform a 4-opt, that does not increase tour length by more than 10%.

3.9 Comparative Performance of Tour Construction and Tour Improvement Operators

When conducting experiments on the TSP one of the difficulties is designing instances of the TSP on which to run the algorithms. Producing random instances is not completely satisfactory because these will not have distributions of real world problems. Another factor with random problems is that no optimal solution or bounds will be known and if the problem is large it may be impossible to find one with the resources available. The TSP library TSPLIB solves the first problem and partially solves the second [74]. TSPLIB contains a collection of contributed problems including city problems, PCB drilling problems and X-ray crystallography. The library contains some solution tours and optimal tour lengths. Many of these problems have appeared in the TSP literature. Using a problem from TSPLIB has the advantage that the results can be compared by other researchers as TSPLIB is freely distributed for research.

In his book [75] Reinelt gives extensive results for all of the algorithms mentioned here using twenty-four euclidean TSPs from TSPLIB. The sizes vary from 198 to 5934 cities. For the best construction techniques, results above 10% of the optimum on big problems are the best that can be expected but these results can be produced in under 10 seconds on a workstation[5] for a 5934 city problem.

If really good results are required, the tour improvement heuristics are required. Node and edge insertion produce rather poor results. If applied to randomly generated tours the results are very poor with tour length on average twice the optimal length. Tours produced by the nearest neighbour heuristic can be improved by approximately 10% and savings tours by 2–3%. The Lin-Kernighan heuristic is the clear winner. Results under 2% are possible in 37 minutes for a 6000 city problem. If a quality compromise to 2.5% is acceptable, this time can be cut to only 5 minutes. The iterated Lin-Kernighan heuristic gives very good results but requires extensive computing time [75].

[5]Sun SPARCstation 10/20

3.10 Data Structures for the Travelling Salesperson Problem

A number of data structures are needed for the representation of the TSP, for the solution, and for the book-keeping during the search. In this section different possible structures are described. Some comments about the resulting storage space complexity of TSP algorithms and heuristics are then made.

3.10.1 Graph Representation

The TSP operates on a weighted complete graph K_n. To represent any graph structure the first step is to have a one-to-one mapping from the set of vertices to the set of integers $\{1, 2, \ldots, n\}$ so that each vertex can be easily represented. So we will assume that such a mapping exists and we can easily move between the representations.

Two of the standard methods for representing graph data structures are the *adjacency matrix* and the *adjacency-structure* [79]. In an adjacency matrix an n-vertex graph is represented by an $n \times n$ matrix c where

$$c_{uv} = \begin{cases} 0 & \text{if } u \text{ is not adjacent to } v \\ 1 & \text{if } u \text{ is adjacent to } v \end{cases}$$

In the adjacency-structure each vertex has an adjacency list that contains all vertices that are adjacent to it.

Either method can be converted for weighted graphs. To represent edge weights in an adjacency matrix let c_{uv} be the weight of the edge uv or 0 if they are not connected. In this case it is called a *weight matrix*. To the adjacency structure it is just necessary to add the weight as an extra field in the adjacency list. The adjacency (or weight) matrix requires n^2 entries while the adjacency-structure varies depending on the density of the graph. Thus the adjacency-structure is more storage-efficient on a sparse graph while the adjacency matrix is better on dense graphs. The TSP operates on a complete graph so it makes sense to use the adjacency matrix because all entries are required. Since we are dealing with a symmetric TSP the weight matrix

is symmetrical so it is possible to reduce storage space by storing only half the matrix. The storage complexity for the adjacency representation remains $O(n^2)$ even if only half the matrix is stored.

On very large graphs it can become very expensive to store a whole weight matrix. For example a 4000-city problem requires almost 64M of memory if a complete matrix is used and each weight is stored as a four byte integer ($4000^2.4$). If the TSP is defined on a metric space with coordinates, such as two or three dimensional euclidean space, then rather than calculate the whole weight matrix at once, distances can be worked out as required. While saving space this will slow an algorithm down if many distance calculations are required and the metric uses a function like square root which is slow in comparison with other arithmetic operations.

3.10.2 Tour Representation

A data structure is needed to represent complete tours or partial tours under construction. The simplest method is to store a permutation as an array. The problem with this method is that it is not possible to find a particular city without a complete scan of the array each time. To reduce these overheads a better representation is an adjacency representation which uses an array where the i-th position in the array contains the successor to city i. It is often also useful to store the predecessor to i as this is useful for working with heuristics like the 2-opt heuristic where paths have to be reversed.

3.10.3 Storage Space Complexity

None of the data structures discussed have exponential space requirements. The tour representation structures have linear space requirements as does the storage of coordinates when working with metric spaces. The weight matrix, which is $O(n^2)$, is the only structure having better than linear space requirements. The heuristics discussed in this chapter also do not have large requirements as they do not need much working storage. One exception is the savings heuristic which can have its time complexity improved from $O(n^3)$ to $O(n^2 \log n)$, but then requires $O(n^2)$ storage space [75].

3.11 Summary

The TSP is a problem of great relevance to both computing and operations research with many real world applications. Although it is an $\mathcal{NP}$-hard problem so that the chance that a 'good' algorithm will ever be found is very remote, there are still a large number of very good heuristic algorithms which exist to solve the problem. The choice of which to use can be made based on the required accuracy and time available. Algorithms like Lin-Kernighan are particularly impressive in the accuracy and speed with which results can be produced.

If such good heuristics already exist for the TSP one may ask why so much research, reviewed in the next chapter, has been devoted to applying genetic algorithms to the TSP. Firstly the TSP has been well established as a testing ground for ideas in combinatorial optimization and it is right that new ideas should be tried on the TSP, as doing so provides insight into both the techniques and the TSP itself. Secondly the nature of the GA is that it can be used to complement existing techniques by using the GA ideas of populations and crossover along with a more traditional heuristic. GAs can be easily implemented on distributed and parallel processors. This means that combining a traditional technique with a GA is one way in which a traditional serial technique can be implemented on parallel hardware. Such hybrids and other techniques will be described in the following chapter. Finally GAs have the flexibility and simplicity to be applied to many problems. The GA may be the perfect choice for a problem that is a TSP with additional constraints that render the traditional techniques described in this chapter unusable.

Chapter 4

Solving the Travelling Salesperson Problem using Genetic Algorithms

4.1 Introduction

Genetic algorithms have been applied to the TSP by a number of authors [12, 15, 36, 47, 43, 63, 69, 87, 90]. Owing to the many different parameters in a GA there is still much scope for research in the use of GAs on the TSP and other combinatorial optimization problems. One reason for this is that GA solutions of the TSP tend to require a number of modifications from the GA presented in Chapter 2 to work efficiently.

Currently, GA-based methods are not competitive with deterministic methods like the iterated Lin-Kernighan (see Section 3.8.5) when time comparisons are made, though very good quality results can be produced [66]. Other methods like simulated annealing have also not yet out-performed these methods [75, ch9]. The main advantage that GA-based (or even simulated annealing) methods have over the more traditional techniques is that they do not require much understanding of the problem structure. This makes it very easy to extend them to problems more complicated than the TSP. This ability to adapt easily to more complex problems is at the expense of performance. If optimizations requiring more domain-specific knowledge are added to the TSP they will make the GA less general-purpose and less easily adapted to other

problems.

This study is to be restricted to the symmetric TSP as was mentioned in Chapter 1 and Chapter 3. Many of the operators described in this chapter will work with the asymmetric TSP. Where this restriction becomes really useful is when the GA is hybridized with standard TSP local optimization heuristics, such as the 2-opt in Chapter 3, which operate only on the symmetric TSP.

This chapter reviews current research on how a GA can be applied to the TSP by reviewing current research. The application of GAs to combinatorial optimization problems poses unique constraints and challenges. The basics needed to produce a GA-based TSP heuristic will first be described. After this, extra steps which add domain-specific knowledge and hybridize the GA with TSP techniques mentioned in Chapter 3 will be described. The implementation and analysis of the results will be covered in the following chapter.

4.2 Applying Genetic Algorithms to the Travelling Salesperson Problem

When applying a GA to any problem the same initial questions discussed in Chapter 2 must be asked:

- What is the fitness function?
- How will the problem parameters be represented (encoded) in a string for manipulation by crossover and mutation?

But there are other choices to be made in the case of combinatorial optimization problems like the TSP. Research has shown that the standard genetic operators of crossover and mutation are ill-suited to this domain [47]. What is really important when choosing a representation is how it performs with the crossover and mutation operators, as these are the only parts of the GA that interact with the representation (apart from the fitness function which decodes it).

It seems then that the choice of representation must be made together with the choice of crossover and mutation operators [83]. Thus the success or failure of the encoding is based on the operators because these will determine how parents are recombined. So two further decisions must be made in conjunction with the representation:

- How should the crossover operator combine parent strings?
- How should the mutation operate on strings?

All of the above questions will be answered in the two sections that follow by firstly looking at a description of the fitness function, and then investigating various representations in conjunction with the crossover and mutation operations that will be used on each.

4.2.1 Fitness Function

The fitness function provides a measure of the fitness of individuals in a GA. In the case of the TSP each individual is a tour. The tour length can be used as a fitness function. As the TSP is a minimization problem we can take the negation of the tour length to get a maximization problem. To obtain positive values we can add a large enough positive scaling factor to make all fitness values positive. So to calculate the fitness function f let $f(x) = t_{max} - t(x)$ where $t(x)$ is the length of tour x and t_{max} is the maximum of all tour lengths in the population of tours.

4.2.2 String Encoding

In the case of the TSP, it is necessary to encode problem solutions into a string. One logical choice is to encode a tour into a string which can be manipulated with crossover and mutation. With a problem like the TSP it makes no sense to encode using a binary alphabet as the smallest unit ever dealt with is a node on the tour. In the literature no attempt is made to use a binary alphabet. Rather, the basic unit is a node.

The choice of how to represent the tour is important in relation to how crossover will combine tours together. One problem that has to be overcome is the possibility of

crossover operating on two parent tours and producing an offspring that is infeasible — no longer a tour. This can be avoided or dealt with in a number of ways [33, 59]:

1. Simply discard any offspring of this nature.
2. Allow infeasible strings into the next generation but apply a penalty function to the fitness function which lowers the fitness depending on how badly the string diverges from being a tour.
3. Design the representation so that this never happens with the standard 1-point crossover operator.
4. Choose a representation with a crossover operator that always produces valid tours.
5. Rather than representing the tour directly, use a representation that suggests how the tour should be constructed, using rules that always produce a valid tour.

Option 1 has not been found very useful in GAs in general [59]. In the TSP it would be particularly inefficient as most offspring generated would be infeasible. Option 2 is often used on GAs operating on problems with constraints [33]. The problem with this approach is that there are so many more infeasible than feasible solutions to the TSP using standard tour representations [59]. If seems that this method is best avoided for the TSP if possible. The other three options will be discussed in the following sections.

Figure 4.1 shows a tour that will be used when giving examples of tour representations.

4.2.2.1 Ordinal Representation

Attempts have been made to design the encoding so that applying the traditional crossover operator always produces valid solutions. If this is a design requirement then a permutation representation cannot be used. One encoding which does work is the *Ordinal Representation* [47].

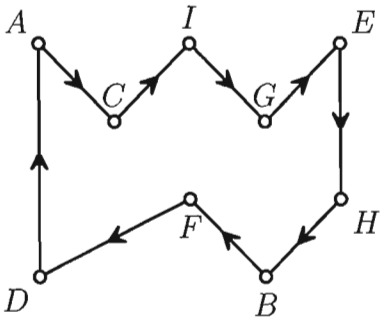


Figure 4.1: Example of a 9-city tour ACIGEHBFD.

To encode the tour into the ordinal representation one starts with a *free list* containing all cities in a canonical order. The position of a city in the free list is appended to the *ordinal list* and this city is deleted from the free list. This process continues until the free list is empty. In order to make the representation unique it is necessary to fix the starting city. Table 4.1 contains an example of the process for Figure 4.1. To decode the result back to a tour start with a full free list and work from the ordinal list back to a tour. It is possible to use standard 1-point crossover and still have a valid ordinal representation as offspring.

Table 4.1: Converting to the ordinal representation

Ordinal List	**Free List**
()	(A B C D E F G H I)
(1)	(B C D E F G H I)
(1 2)	(B D E F G H I)
(1 2 7)	(B D E F G H)
(1 2 7 5)	(B D E F H)
(1 2 7 5 3)	(B D F H)
(1 2 7 5 3 4)	(B D F)
(1 2 7 5 3 4 1)	(D F)
(1 2 7 5 3 4 1 2)	(D)
(1 2 7 5 3 4 1 2 1)	()

This approach has been shown to produce very poor performance that is no better than that of random search [47] because it does not combine good subtours into

the offspring. When two parents are combined the result bears very little resemblance to the parents. It could be said that this representation does not adhere to the *building block hypothesis* since very small changes to a string result in large alterations to the tour.

4.2.2.2 Path Representation

A natural way to encode graphs into a string is to use a permutation to represent a path. This is a choice used widely in the literature [36, 43, 69]. For each tour of length n there are n different ways to represent the tour depending on the starting city. If this is not desirable it is possible to fix the starting city. In this representation the absolute positions of cities in the string are not significant. It is the relation of each symbol to the next that actually encodes the phenotype. One consequence of this approach is that the standard theoretical backing for GAs described in Chapter 2 does not apply. This has led to the development of an alternative definition of schemata called *o-schemata* [36]. In this encoding,

(A C I G E H B F D)

represents the tour in Figure 4.1.

4.2.2.3 Adjacency Representation

In the *adjacency representation* a gene j at locus i represents the edge in the tour from city i to city j [47]. Using this encoding it is possible to represent a sequence of edges that does not form a tour. For this reason it is not possible to use traditional crossover. One feature of this representation is that it is unique for a given tour. In this encoding,

(C F I A H D E B G)

represents the tour in Figure 4.1. The first symbol indicates an edge in the tour from city A to city C, the second an edge from city B to city F, the third an edge from city C to city I, and so on. As with the path representation the genes are not independent of each other because each city can appear only once in this representation.

4.2.2.4 Representation Comparison

All the representations given here can be used for any problem that needs to represent permutations. These representations are not only useful in the TSP, but also in other problems where permutations are useful, for example, in scheduling problems [85].

Of the three representations given here the ordinal representation has already been shown to be useless for the TSP using standard crossover operators. As this could really be the only motivation for using it, it will not be considered further.

Both the path and adjacency representations require special-purpose crossover and mutation. The primary difference between the representations is that in using the path representation there is a particular starting city while the adjacency representation does not have a starting city. The choice of which to use is based on which representation provides the easier implementation of the operators. The next section looks at crossover operators. Some try to preserve adjacency information while others try to preserve relative order.

4.2.3 Crossover Sequencing Operators

In this section we look at crossover operators designed for operating on permutations (whether they are represented using path or adjacency representations does not matter). These operators take parent permutations and always produce permutation offspring. They are commonly called sequencing operators [82]. A number of different approaches exist for the choice of the crossover operator. In the case of the TSP, adjacency information is what is important so it would be expected that crossover operators that best preserve adjacency information will have a better chance of success. The operator has to balance the need to preserve the adjacency information present in the parents with the need to generate offspring that are different from the parents [27].

In the following discussion of crossover operators we will assume two parent strings:

$$A = a_1 a_2 a_3 \ldots$$

$$B = b_1 b_2 b_3 \ldots$$

4.2.3.1 Order Crossover

Order crossover described here is a modification of Davis's ordered crossover [17]. Two crossover points are selected, and the string between the two points in the first parent is copied whole to the offspring. Nodes are then copied from the second parent, starting at the second crossover point. Any nodes already incorporated from the first parent are ignored. The second offspring is derived in the same way but with the parents swapped. The only difference from Davis's crossover is that in Davis's method only the second crossover point is selected, the first always being the start of the string [69]. It is possible to generalize this approach by dividing the string multiple times [12]. See Table 4.2 for an example of the order crossover operator. The boxed sections in the parents denote the sections selected by the crossover points. Figure 4.2 shows order crossover applied graphically to two parent tours.

Table 4.2: Order Crossover Examples

Parent 1	Parent 2	Offspring 1	Offspring 2
ACI DBELHKGJ F	AIJ DGECBKFL H	AICFDBELHKGJ	AIHJDGECBKFL
AF LIGK HCJBDE	AK CDLH IBFGJE	ACDHLIGKBFJE	AFIGKCDLHJBE
AGIJ HLF KECBD	AIDB CGH KELFJ	AIDBCGHLFKEJ	AIJLFCGHKEBD
A IBGDCLJF KEH	A KEFCBJLD IHG	AKEIBGDCLJFH	AIGKEFCBJLDH
ALJD FGBEICHK	ABKJ DHECFLGI	AJDLFGBEICHK	AJBKDHECFLGI

4.2.3.2 Partially Mapped Crossover

Partially mapped crossover (PMX) was developed by Goldberg and Lingle [36]. It also makes use of two crossover points. The section of the string demarcated by the points is called the *mapping section.* This is used to define a swapping of nodes. So if the mapping section is $a_i \ldots a_j$ and $b_i \ldots b_j$ then we define a mapping from A to B as a_k swaps with b_k for $i \leq k \leq j$. Other nodes map to themselves. This gives one offspring derived from A using the mapping, and one offspring derived from B using the mapping [69]. See Table 4.3 for an example of the partially mapped crossover operator. The boxed sections in the parents denote the sections selected by the crossover points.

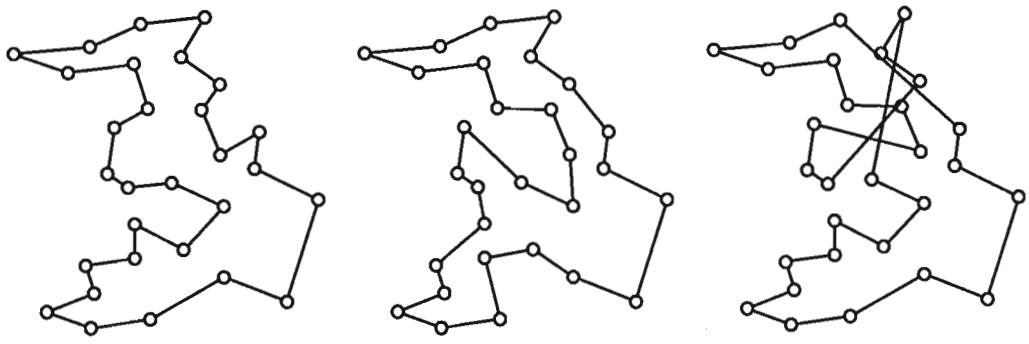


Figure 4.2: Order Crossover In Action

Figure 4.3 shows PMX applied graphically to two parent tours.

Looking at both the figure and the table it can be seen that PMX partially preserves adjacency information and partially preserves positional information. Since adjacency information is what is important for the TSP one cannot expect great performance from PMX.

Table 4.3: Partially Mapped Crossover Examples

Parent 1	**Parent 2**	**Offspring 1**	**Offspring 2**
ACI DBELHKGJ F	AIJ DGECBKFL H	AJIDGECBKFLH	AICDBELHKGJF
AF LIGK HCJBDE	AK CDLH IBFGJE	AFCDLHKGJBIE	AHLIGKDBFCJE
AGIJ HLF KECBD	AIDB CGH KELFJ	ALIJCGHKEFBD	AIDBHLFKEGCJ
A IBGDCLJF KEH	A KEFCBJLD IHG	AKEFCBJLDIGH	AIBGDCLJFKHE
ALJD FGBEICHK	ABKJ DHECFLGI	ABJKDHECFLGI	ALDJFGBEICHK

4.2.3.3 Cycle Crossover

Cycle crossover [69] is defined to meet the following conditions:

- Every position must retain a value present in one of the parents.
- It should, in general, be different from both parents.

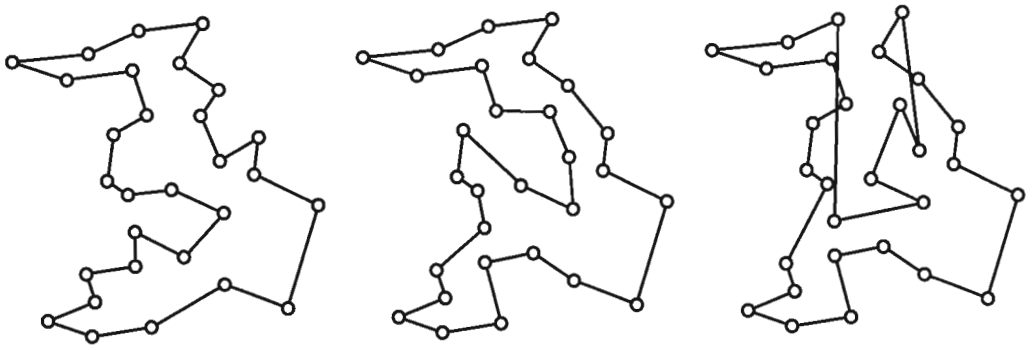


Figure 4.3: Partially Mapped Crossover In Action

- As usual, it must be a permutation.

The algorithm for cycle crossover, producing an offspring from parent strings A and B, is presented in Algorithm 6 [82].

Algorithm 6 (Cycle Crossover Algorithm)

Select at random an element $a = a_i$ of A.

REPEAT

Copy a_i into the offspring at position i.

Let $i = j$ where $b_j = a_i$.

UNTIL $a_i = a$.

For each position not yet filled in the offspring, copy the element from the corresponding position in parent B.

■

4.2.3.4 Edge Recombination

The original *edge recombination* operator was introduced by Whitley et al. (see [90]). The primary aim of the edge recombination operator is to introduce as few edges as possible that do not exist in the parent tours.

The edge recombination operator builds an *edge table*. An edge table is functionally the same as the adjacency-structure described in Section 3.10.1. For each city it lists the cities adjacent to it in either of the parent tours. Each city is adjacent to at least two cities and at most four. A new tour is now constructed using Algorithm 7 [91, 63]:

Algorithm 7 (Edge Recombination)

Assume the graph has n cities. Randomly select the initial city a_1.

Let i = 1.

REPEAT

Remove a_i from every city's adjacency list.

IF there is a city adjacent to a_i

THEN

let a_{i+1} be the city with the shortest adjacency list, breaking ties randomly.

ELSE

let a_{i+1} be a random city not in $a_1, a_2, \cdots, a_i$.

END IF

Let $i = i + 1$.

UNTIL $i \geq n$.

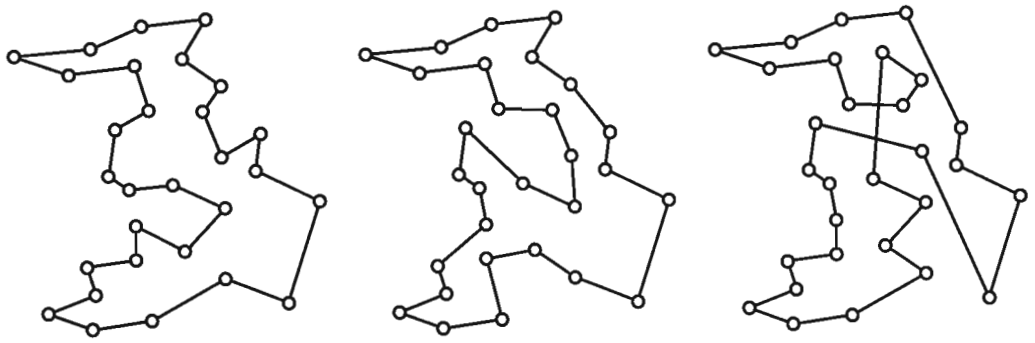
Permutation $(a_1, a_2, \cdots, a_n)$ is the offspring tour.

■

Table 4.4 illustrates the behaviour of the edge recombination operator. The superscript 1 marks the first city chosen randomly as a starting point. Subsequent superscripts mark cities chosen at random when no other choice was available. Figure 4.4 shows a graphical view of edge recombination.

Table 4.4: Edge Recombination Examples

Parent 1	**Parent 2**	**Offspring**
AEFDKLHJIBCG	ALEDKIHJCGFB	AC^{1}GFDELKIJHB^2
AFCIKBLHEDJG	AHCJIKBLEFGD	A^{2}GDI^{1}JCFEHLBK
ALKCEFGJBHDI	AGBCKDELFHIJ	AH^{2}DE^{1}CKLFGBJI
AEBFJKHICLDG	ALKGHJEBIDCF	AI^{1}BEJHKGDLCF
ABILDHJGCEKF	AECBKJHLGIFD	A^{1}FKECBIGJHDL



Parent 1 Parent 2 Offspring

Figure 4.4: Edge Recombination In Action

Edge recombination works on the principle that there should be as little disruption as possible to the parent tours. As any introduction of edges not currently in the parents can extend the tour length without limit it is prudent to avoid the introduction of such edges. Edges in an offspring tour that existed in neither parent tour are called *foreign edges*. The edge recombination operator tries to avoid these foreign edges. It is, however, in general unavoidable to introduce at least one foreign edge because the final edge used to return to the home city is chosen implicitly.

Whitley has been able to demonstrate that this operator manipulates an underlying binary encoding. This makes it possible to apply some of the GA theory, much of which is developed for binary encodings (see Section 2.5), to this operator [91]. A number of improvements to the edge recombination operator have been suggested, some of which can be motivated by GA theory. A number of changes and

improvements are described in the next few sections.

4.2.3.5 Edge-2 Recombination

Edge-2 recombination is an enhancement to the original edge recombination operator. It aims to improve on edge recombination by favouring edges that are in both parents [82, 63]. If a city is adjacent to the same city in both parent tours then it is specially flagged by negating the city number in the edge table. When looking for the next edge a flagged city is preferably selected. If there are no flagged cities in a city's adjacency list then edge-2 recombination reverts to the selection rules used for edge recombination in Algorithm 7. Table 4.5 illustrates the behaviour of edge-2 recombination. The superscript 1 marks the first city chosen randomly as a starting point. Other superscripts mark cities which were chosen at random when no other choice was available. Figure 4.5 shows a graphical view of edge-2 recombination.

Table 4.5: Edge-2 Recombination Examples

Parent 1	**Parent 2**	**Offspring**
AGKLEIFDBCJH	AHDIKGFEJCLB	ABCJEFIKGL^{2}D^{1}H
AJDCEFLIGBHK	AJFKLECHBDGI	AKH^{1}BGILFECDJ
ACJFEGBHDLKI	AHJKILFDCGEB	ABEGF^{2}I^{1}KLDHJC
AGCBFHLIJKDE	AILFHGDJCEKB	AE^{1}DKJILHFBCG
ADCELHJFKGBI	AHFDGICEJLKB	AHJFDG^{1}KLECIB

4.2.3.6 Edge-3 Recombination

The term *terminal* is used to describe a city at either end of a partial tour, where all edges in the partial tour are inherited from the parents. A terminal is said to be *live* if it still has edges in its edge list; otherwise it is said to *dead*. The *edge-3 recombination* operator [63] improves on edge-2 recombination as follows. When the partial tour reaches a dead terminal, instead of continuing with a random city, it reverses the partial tour and continues it from the other terminal (if this is possible, that is, if it still live). When the next failure occurs, both ends of the tour segment are dead

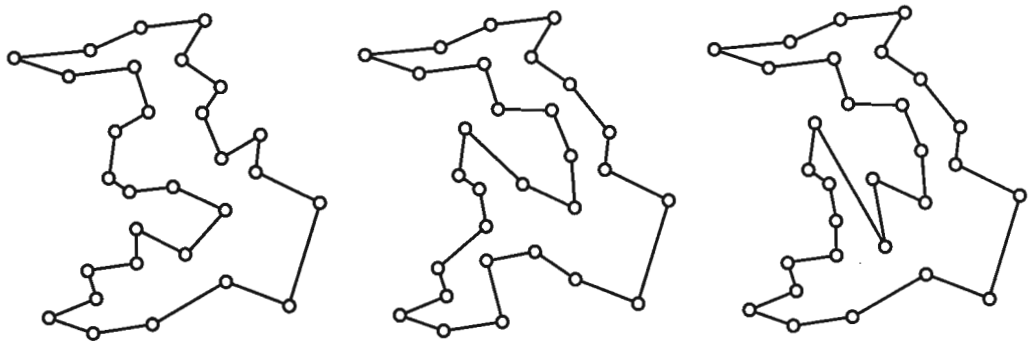


Figure 4.5: Edge-2 Recombination In Action

terminals so it is necessary to select a random city to start a new partial tour. This partial tour is constructed in the same way, and the process continues until a complete tour is obtained.

The edge-3 recombination operator introduces fewer foreign edges then edge-2 since failures which require the random selection of an edge should, on average, occur less often.

4.2.3.7 Edge-4 Recombination

Edge-4 recombination extends the idea used in edge-3 recombination in trying to reduce the number of foreign edges introduced [27]. Edge-4 proceeds as for edge-3 until both ends of the partial tour are dead terminals. An attempt is then made to reverse a segment of the partial tour to get a live terminal so that the partial tour can be continued. Say the partial tour under consideration is $(x_1, x_2, \ldots, x_i)$ and the cities adjacent to x_i in the edge list are $X = \{x_{a_1}, x_{a_2}, \ldots, x_{a_j}\}$. These are all part of the partial tour since x_i is a dead terminal. Let Y be the set of cites that follow the cities of X in the partial tour ie $Y = \{x_{a_1+1}, x_{a_2+1}, \ldots x_{a_j+1}\}$. The city with the fewest cities left in its edge list is selected, say x_k. The partial tour from x_k to x_i is then reversed so that x_k is now the terminal city. If x_k still has edges in its edge list the tour is continued from there. If not, the partial tour is reverted to its original ordering and

another city in Y is considered. If this process fails then the other terminal city is tried. If this still fails then a random city is selected to start a new partial tour, as is the case in edge-3 recombination.

The edge-4 operator does perform slightly better than edge-3 [27]. It has the disadvantage that it is slower because of all the backtracking and more complex to implement. This could mean that it offers no real advantages over edge-3 [27].

4.2.3.8 Maximal Preservation Crossover

The *Maximal Preservation Crossover* (MPX) operator is somewhat similar to order crossover (Section 4.2.3.1). It has been successfully used to solve TSPs by Eshelman [29] and Muhlenbein [66, 67]. This description is based on Muhlenbein. MPX operates on two parents, the *donor* and the *receiver*. A section is copied directly from the donor into the corresponding position in the offspring. The rest of the genes are filled in by copying consecutively from the receiver. The rules in Algorithm 8 cover the cases where simple copying will create an illegal tour. The length k of the initial segment is chosen between bounds b_{low} and b_{up}. It has been found in some empirical studies that a fixed value for the length of the initial segment gives better performance [63]. A fixed value of one third of the tour length is suggested for the initial segment in [87]. See Table 4.6 for examples of MPX in operation. Figure 4.6 shows a graphical view of MPX. It is interesting to compare this figure with the illustration of PMX (see Figure 4.3) which is clearly more disruptive.

Algorithm 8 (Maximal Preservation Crossover)

Assume the tour is of length n and we have a donor tour

$$\{v_{x_1}v_{x_2}, v_{x_2}v_{x_3}, \ldots, v_{x_n}v_{x_1}\}$$

and a receiver tour

$$\{v_{y_1}v_{y_2}, v_{y_2}v_{y_3}, \ldots, v_{y_n}v_{y_1}\}$$

Choose i and k randomly such that $0 \leq i < n$ and $b_{low} \leq k \leq b_{up}$.

Let $z_j = x_j$ for $j = i, i+1, \ldots, (i+k)$ MOD n.

Assign successive edges $v_{z_{p-1}}v_{z_p}$ for $p = (i+k+1), \ldots$ until a tour is built using the following priority scheme:

1. Let $z_p = y_i$ where $v_{z_{p-1}}v_{y_i}$ is an edge in the receiver and if it will not violate tour conditions.
2. Let $z_p = x_i$ where $v_{z_{p-1}}v_{x_i}$ is an edge in the donor and if it will not violate tour conditions.
3. Let $v_{z_{p-1}}v_{z_p}$ be the next available edge in the receiver that does not violate tour conditions. This is viewed as implicit mutation.

The tour $\{v_{z_1}v_{z_2}, v_{z_2}v_{z_3}, \ldots, v_{z_n}v_{z_1}\}$ is the resulting offspring.

■

Table 4.6: Maximal Preservation Crossover Examples

Donar	Receiver	Offspring
AE FDKL HJIBCG	ALEDKIHJCGFB	ABFDKLEIHJCG
AEB CIJH FDLKG	AFHKEGBDJLCI	AFCIJHKEGBDL
AGHIC FBJD LEK	AJKHGEFICDLB	AFBJDLEKHGIC
AF HBIJEDKG CL	AFDGHICLKBEJ	AFDGHIJEKBCL
AL IFGBKCHJ DE	AFJLGHCKDBIE	ALGHCKBIFJDE

4.2.3.9 Performance Comparison

Oliver, Smith and Holland [69] used a 30 city problem to compare order crossover, PMX and cycle crossover. Their experiments showed that order crossover performed best, followed by PMX. Cycle crossover performed the worst. A study in [82] gave the same ordering for these operators but also compared the edge-2 operator. The results showed that edge-2 is superior to the other three operators. Two other operators designed for schedule optimization from [85] were also compared — another order crossover (referred to as order crossover #2) and a position-based crossover. Results for these operators lay between those of order crossover and PMX.

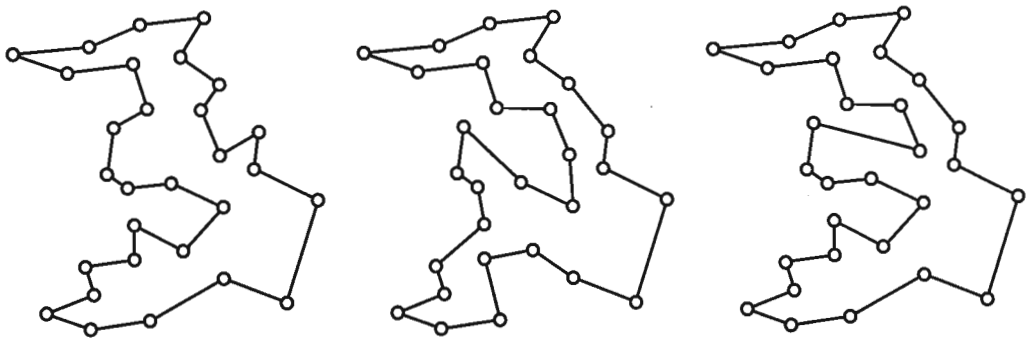


Figure 4.6: Maximal Preservation Crossover In Action

A further study, using certain metrics to measure the correlation between parent and offspring fitness together with experimental results indicated that edge recombination is best (they did not test edge-2), followed by order crossover, PMX and finally cycle crossover [62]. Making use of these and other metrics and experimental results, [63] shows that edge-3 performs better than edge-2 which performs better than MPX. However, if local optimization (2-opt) is used then MPX performs better than both of the edge operators. Further studies, including edge-4 recombination and all the other operators mentioned, found similar results — that the edge family are comparable to MPX when no local optimization is used but that MPX moves ahead when used with local optimization. Edge-4, while performing slightly better than edge-3, requires greater CPU time and may not be worth the expense [27].

4.2.4 Mutation Operators

This section looks at operators that can perform the function of mutation in the standard GA. It has already become clear that a GA for solving the TSP is rather different from a standard GA. Just as the representation requires that the crossover operator preserve the tour, so must the mutation operator do the same.

4.2.4.1 Node Insertion

An easy way to implement a mutation operator is to exchange the positions of two cities in the permutation. This is one of the tour improvement moves described in Chapter 3 Section 3.8.1. This is easy to implement and corresponds closely with mutation in the general GA. This operator performs poorly, however [82]. This is a similar result to that discussed in Chapter 3 where node insertion is a poor tour improvement operator (but useful when used together with 2-opt).

4.2.4.2 Remove and Reinsert

Two nodes are selected and one is removed and reinserted after the other. This operator performs better than node insertion [82].

4.2.4.3 r-Opt Moves

In Chapter 3 various heuristic TSP algorithms were seen to make use of what are known as *r-opt* moves, where r edges in a tour are replaced by r different edges to produce a new, shorter tour. This same technique can be used for a mutation operator, and is a logical choice of operator for the TSP domain. In the case of the commonly used 2-opt move we are just reversing a path within the tour. Thus the operation being performed is really the inversion operation seen in Section 2.7.6. In the case of the TSP, where relative position is important, it would seem that the inversion operator makes some sense, and indeed this operator performs better than both the previous operators [82].

4.2.5 Redundant Codings

A tour can be encoded using a *redundant coding*, which is self organizing. Within a redundant coding, some information in the string may be unused. Some may be missing from it. Some scheme has to be designed to convert the coding into a valid tour by deciding which parts of the code to ignore and what defaults to use when information is missing.

A number of different redundant codings have been devised along with corresponding interpretations. Two are given here. One encoding is based on the ordering in the tour while the other tracks adjacency information.

4.2.5.1 Redundant Order Representation

An n-city tour is encoded in a string of n (city, tour order) pairs. The pair is known as a *macro-gene* and the city and sequence in the tour as *codons*. In this encoding,

E5 F8 B7 D9 G4 I3 H6 C2 A1

represents the tour in Figure 4.1. It is necessary to fix a home city as the start of the tour [86].

In order to transform this redundant coding into a tour, three problems have to be dealt with — cities which have duplicate ordering information, positions that are occupied by more than one city, and cities which have no position information at all. The following rules are used to remove ambiguity and complete missing information:

1. If a city is represented in more than one macro-gene then select one of the macro-genes randomly.

2. If the same position is represented in more than one macro-gene then select one of the macro-genes randomly.

3. If after all macro-genes have been interpreted there are cities that have not been positioned then randomly place them in unused positions in the tour.

4.2.5.2 Redundant Edge Representation

An n-city tour is encoded in a string of n macro-genes which consist of two codons, both of which represent a city. Thus the macro-gene represents an edge in the tour. In this encoding,

EH CI DF IG CA GE HB FB DA

represents the tour in Figure 4.1. It is necessary to fix a home city as the start of the tour [86].

A redundant edge representation may also have a number of problems that need correcting. Algorithm 9 [86] can be used to produce a tour from this representation. It removes ambiguity by selecting left-most macro-genes when there is a choice and choosing random cities to fill in the tour when it is incomplete.

Algorithm 9 (Interpret Redundant Edge Representation)
Let $S = (A_1B_1, A_2B_2, \ldots, A_nB_n)$ be the sequence of macro-genes.

Let u_1 be the first codon in the first macro-gene and let $i = 1$.

WHILE $i \leq n$

If city u_i is contained in a single macro-gene of S then set u_{i+1} equal to the other codon of that macro-gene. Remove the macro-gene containing u_i from S.

If city u_i is contained in more than one macro-gene of S then set u_{i+1} equal to the other codon of the left-most macro-gene. Remove any macro-genes containing u_i from S.

If city u_i is not contained in any of the macro-genes of S then set u_{i+1} equal to a random city z such that $z \neq u_j$ for $j = 1, 2, \ldots, i$.

Let $i = i + 1$.

END WHILE

■

4.2.5.3 Comparison of Redundant Codings

Comparison of these two codings with 1-point crossover by Tamaki has shown that the redundant edge representation performs best even when compared against some variations of the redundant order representation. For problems of about 64 cities it was necessary to enhance the implementation by including some local search when constructing a phenotype from the genotype. Lamarckian inheritance (see Section 2.1.1), where the genotype is modified during reproduction to better represent the phenotype produced from the redundant representation, was also used.

4.3 Hybrid Genetic Algorithms for The Travelling Salesperson Problem

The version of the TSP dealt with so far in this chapter is actually a more difficult version of the TSP originally described in Chapter 3. It is called the *blind* travelling salesperson problem, because only the length of the entire tour is used, not the lengths of the individual edges [33, p170]. Although GAs have been applied successfully to the TSP, they have performed well on large problems only when the GA is enhanced with other techniques [87]. In particular it has been shown that it is worthwhile to make use of local search heuristics because these improve both the quality of solutions and the speed with which they are produced [66, 87]. This section will cover a variety of techniques which incorporate additional problem-specific knowledge into the GA.

4.3.1 Population Initialization

Population initialization has been discussed as a method to improve the speed and results of a GA. In the case of the TSP, any of the tour construction heuristics in Chapter 3 can be considered. An obvious requirement is that the heuristic should be fast with respect to the results it produces. Often, initializing the population will result in quicker performance of the GA, but this is not useful if the initialization itself takes a long time in comparison to the time taken by the GA.

An important aspect of initializing the population is that it should contain a good distribution of allele values to be used as building blocks [43]. In the extreme case, it is useless to initialize the whole population with exactly the same tour. Many of the algorithms in Chapter 3 produce the same or similar tours each time they are run, which makes them unsuitable.

The nearest neighbour heuristic described in Section 3.7.1 produces different tours depending on the starting city, and runs in $O(n^2)$. This can be improved on, as described in Chapter 3, using candidate subgraphs. The question is — how different will these tours be if the whole population is initialized using this method? A probabilistic version of the nearest neighbour heuristic operates by selecting the closest city

of a random sample of remaining cities (the sample size may be fixed). This heuristic produces tours that are better than average tours, but when it is used to initialize a population, the result is more varied than that of the ordinary nearest neighbour heuristic. It also has the advantage of running in $O(n)$ [43]. Grefenstette compared the performance of these two initialization methods, showing that while initializing with the nearest neighbour produces very good results initially, the GA cannot produce much improvement on these results because of the high allele loss [43]. The probabilistic nearest neighbour heuristic produces better final results but takes some time to converge. In contrast to this Chatterjee *et al* found that using the nearest neighbour heuristic to initialize a population had no noticeable effect on convergence [15]. The different results could be explained by their different structures — the Chatterjee *et al* GA was entirely mutation based.

4.3.1.1 Heuristic Crossover

Heuristic crossover differs from all the other operators discussed here because it uses the edge weight information during offspring construction to select cities [47]. In one form of the heuristic crossover, a random city is selected and the two edges leaving that city in the parents are considered and the shortest one selected. If a cycle would be introduced by selecting the shortest edge then a random edge is chosen. This process is continued until a tour is completed [47].

One criticism of this method is that it evaluates the worth of sections of the tour before the offspring has been produced, which has no biological motivation. Introducing a greedy algorithm like this in a GA could limit the robustness of the GA [30]. The real test, however, is in the results produced by methods like this.

A general class of heuristic crossover operators can be created using Algorithm 10 [43].

Algorithm 10 (Heuristic Crossover)

Randomly select a starting city v_1. Let $i = 1$.

WHILE $i \leq n$

Consider the four cities adjacent to v_i in the parents. Define a probability distribution over the four cities such that a visited city has probability zero.

Select a city z based on this distribution or choose z randomly from among the unvisited cities if v_i is not adjacent to any unvisited cities in the parents.

Let $v_{i+1} = z$.

Let $i = i + 1$.

END WHILE

■

If e_1, e_2, e_3, e_4 are the edges under consideration for selection, the probability p_i that edge e_i will be selected in the above algorithm can be calculated in a number of ways [43]:

1. Assign probability 1 to the shortest edge. In this case we have the crossover operator introduced at the beginning of the section.

2. Assign a uniform distribution across all edges. In this case the weights of edges are ignored.

3. Bias the choice in favour of shortest edges:

$$p_i = \frac{1}{c_{e_i} \sum_{j=1}^{4} \frac{1}{c_{e_j}}}$$

where c_{e_i} is the weight of edge e_i.

4. Combine the above two crossovers by adapting the probabilities so that probabilities start off biased in favour of short edges, but as the population approaches uniformity the distribution approaches that of a uniform distribution.

4.3.2 Local Improvement Operators

It has been observed that GAs can locate good areas of the search space quickly but are not as good at local optimization [43]. Local search operators, like those

studied in Chapter 3, can be applied to problems like the TSP. Results from other researchers have shown that it is fruitful to spend much of the GA running time on local optimization [66, 87].

Some researchers use local search operators in place of mutation [12]. Mutation is generally credited with performing local optimization in the standard GA, among other tasks. So it is natural that mutation can be replaced with a more efficient and direct form of local optimization. Mutation is also intended to help the GA escape from a local minimum. Replacing mutation with local optimization means that this function is no longer performed by the mutation operator. However, crossover operators for GAs solving the TSP generally result in some mutation (the introduction of foreign edges) in offspring tours [91] so the loss is balanced to some extent.

Once a local optimization method has been selected it can be applied in a number of ways. Local optimization can be applied to all individuals so that the GA effectively always operates on locally optimal strings [66]. In this case, the initial population and any subsequent offspring or mutated strings need to have the optimization operator applied to them, and the local improvement will dominate the running time. To save time, the operators can be applied only to certain generations [63]. The other alternative is to apply the local improvement operator at a fixed or adaptive probability, as is done with mutation and crossover. In the next few sections different local improvement operators are considered.

4.3.2.1 Node Insertion Local Improvement Operator

Node insertion is exactly as described in Section 3.8.1. The tour is improved by checking that each node is in the best position in the tour. Since it takes time $O(n^2)$ to check all nodes it is an expensive operation. It is also not a very good local search method as mentioned in Section 3.9.

4.3.2.2 r-Opt Local Improvement Operators

The r-opt improvement described in Section 3.8.4 can be applied to tours as a local improvement operator. Generally the 2-opt is used [29, 63, 67] as it is cheapest. A

single rather than complete 2-opt pass can be used to reduce the amount of time required for local optimization [63, 67].

4.3.2.3 Generalized Lin-Kernighan r-opt

The adaptive r-opt procedure in the Lin-Kernighan heuristic (see Section 3.8.5) is a powerful local search method. Some researchers have found it worthwhile to use this rather expensive heuristic as a local search operator in a GA [87].

4.3.2.4 Repair Local Improvement Operator

The repair improvement operator is a selective form of other local improvement methods designed to reduce the running time required. It is based on the observation that the crossover operator often disrupts only some of the cities, as does mutation. If a tour is locally optimal before crossover and mutation are applied, then it may be possible to repair the tour by considering for local improvement only those cities that have altered positions in the offspring [12, 63].

4.4 Summary

The travelling salesperson problem has been a popular problem on which to test new ideas. Consequently, many researchers have applied the genetic algorithm to the TSP. Interestingly, many different approaches have been attempted and many are successful. This chapter has looked at the many techniques that can be applied. In the following chapter some of these different techniques will be tested and compared against other traditional methods to determine their effectiveness.

Chapter 5

Experimental Methodology and Results

5.1 Introduction

The previous chapter looked at how GAs can be applied to the TSP. This chapter will present an implementation of a TSP-solving GA and the results obtained from it. The experiments that were run and the procedures used will also be described.

A large number of parameters are available in a GA and this is particularly true of a GA designed to solve the TSP where there is less guidance for the choice of some of the parameters because of the limited amount of theory in the area. By their nature, TSP-solving GAs also require decisions about the choice of crossover and local optimization function to use. This research attempts to investigate some of these issues.

Another aspect in this work is the comparison with other GA-based solutions to the TSP. A number of results were reviewed concerning the TSP to compare the results that had been obtained. In addition to the TSP solutions, results that can be obtained with the traditional Lin-Kernighan TSP heuristics were also reviewed to place GA performance in some context. The Lin-Kernighan algorithm, which was mentioned in Chapter 3 as one of the best heuristics for the TSP, was contrasted against the GA approach.

5.2 Genetic Algorithm Implementation

The GA used for the experiments was implemented in C++ using the GNU compiler gcc[1] and the GNU C++ library[2]. The programs assume that all distances are integers and that the total tour distance can be represented in a 32-bit integer (a C `long int` on most 32-bit platforms). This was for compatibility with TSPLIB [74], the library of TSP problems compiled and maintained by Reinelt. TSPLIB problems are all based on integer distances.

All test problems were taken from TSPLIB. Internally all TSP problems were represented as a weight-matrix. If necessary, TSPLIB problems were converted to this form when they were loaded. This conversion was necessary for the problems that are defined as points in some metric space, for example points in euclidean space. The advantage of loading this data into a weight-matrix, rather than doing the calculation, as the distances are needed, is that the distance calculations can be expensive and so it is preferable not to do the calculation for each pair of cities more than once.

The programs were designed to form a working environment which could be used to test different genetic operators, insertion methods and local search methods by providing different parameters. It was also made easy to add new operators, so that a number of different strategies could be tried for each individual run. The GA designed can only be used to solve TSPs but is modular enough to be modified into a general GA framework.

The basic design requirement was flexibility to experiment easily with different settings. For this reason a decision was taken to allow binding of particular choices to happen at run time rather than at compile time. In particular, C++ virtual functions were used. This provided for an environment in which different choices could be tried without recompiling but resulted in some loss of speed because of the extra expense of late binding and the additional code required to support options which might not be used in a particular run. For this reason, running times could be reduced for a particular set of parameter settings if the code were optimized for only those settings. However, as indicated above, this disadvantage was outweighed by the

[1] gcc version 2.7.2.

[2] GNU C++ library version 2.7.1.

flexibility achieved in the variety of experiments that could be performed.

5.2.1 Genetic Operators

In most genetic algorithms crossover and mutation are treated as very different operators. First crossover is performed, and then this is followed by mutation on the result of the crossover. This is how Goldberg's simple genetic algorithm is implemented [33]. This is illustrated in Figure 5.1. In this implementation the only difference between a

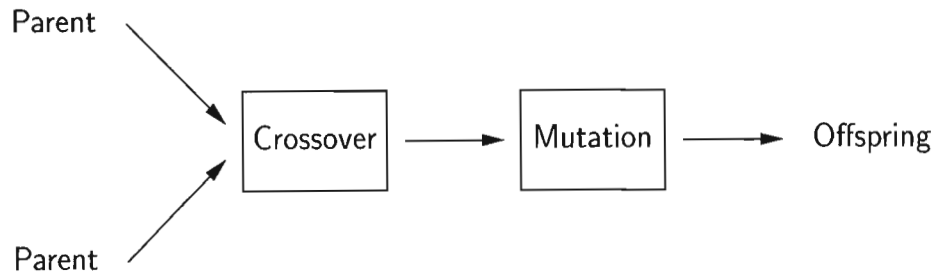


Figure 5.1: Standard Crossover followed by Mutation

crossover and a mutation operator is the number of strings on which they operate. A mutation-like operator is defined as an operator that takes a single string as input and produces a single string as output. A crossover-like operator is defined as an operator that takes two strings as input and produces a single string as output. Apart from the number of input strings these operators are treated in the same way. For each recombination only one of these operators is applied.

There were two reasons for defining the crossover operator to return only one offspring. Firstly, it is quite common in GA implementations for crossover operators to return only a single offspring, for example GENITOR [82]. In particular, TSP crossover operators such as the edge family and MPX return only a single offspring. Secondly, it makes the design of a GA easier if mutation and crossover both insert only one offspring back into the population because they can be treated in a similar manner.

Each reproductive step involved the selection not only of parents but also of a single operator to apply. The offspring produced was always the product of a single operator unlike some GAs where crossover is applied, followed by mutation on the

offspring. Because of the design, a combination of genetic operators could be applied during a run, with selection based on various criteria, including fixed probabilities, scaled probabilities based on run time, and adaptive probabilities based on performance. This also meant that experiments could be performed using:

- only mutation,
- only crossover,
- both mutation and crossover, or
- multiple mutation and crossover operators.

5.2.2 Stopping Conditions

Unlike some heuristics, such as Lin-Kernighan, there is no clear stopping condition for GAs when applied to TSPs. In all these experiments the GA was run for a fixed number of recombinations that was decided on beforehand. The number of recombinations was decided on by looking at other research, and by experimentation.

Alternative approaches, which were not used, are to stop the run when no more progress is being made or to use known optimal, upper bound or lower bound values obtained elsewhere. Although this type of stopping condition sounds like a cheat it can be used practically to solve problems where an upper bound has been found for the particular problem via some other method and a GA is being used to attempt to find a better upper bound.

5.2.3 Parameter Values

Apart from the crossover, mutation and local improvement operators, various other parameters were experimented with, such as population size. The full list of parameters is shown in Table 5.1. The variable p represents the population size. This table lists some of the parameter values using abbreviations which are explained below.

Table 5.1: GA Parameters

Parameter	**Value**
Generation Gap	$\{x : x \geq 1/p \text{ and } x \leq 1\}$
Population Initialization	NN, SNN, Random
Initial Population Improvement	2-Opt, none
Insert Method	Uniform, Average, Worst, Exponentially
Local Improvement	none, 2-Opt
No Duplicates	false, true
Operators	PMX, Over, Edge, Edge-2, Edge-3, MPX
Population Size	
Problem	
Rank Selection	false, true
Max Recombinations	
Sample Method	Universal, Roulette, Random

5.2.3.1 Generation Gap

The generation gap G indicates what fraction of strings are replaced in each generation. In a population of size p, then, Gp is the number of strings replaced each generation. If $G = 1$ then the whole population is replaced each generation. If $G < 1$ then the GA is a steady-state GA which was described in more detail in Section 2.7.3.

5.2.3.2 Insert Method

The insert method was the method used to replace the old strings with new ones for steady-state GAs. Methods used include:

- **uniform:** replacing strings randomly;
- **worst:** replacing worst strings;
- **average:** replacing randomly but only in the worst half of the population;
- **exponentially:** replacing randomly with a probability that decreased exponentially from best to worst strings.

For more information on insertion see Section 2.7.3.

5.2.3.3 Handling of Duplicates

Steady-state GAs actually insert the new offspring into the population. For this reason it is possible to decide not to insert a fit individual. One reason for this choice may be because it is too close, according to some measure, to existing individuals in the population. In this implementation it was made possible to exclude the insertion of duplicates into the population.

5.2.3.4 Crossover Operators

A number of crossover operators were experimented with. They are listed in Table 5.2. These operators were fully described in Section 4.2.3. The operators edge-4, cycle

Table 5.2: Crossover Operators Used in Experiments

Abbreviation	**Description**
Edge	Edge Recombination
Edge-2	Edge-2 Recombination
Edge-3	Edge-3 Recombination
MPX	Maximal Preservation Crossover
PMX	Partially Mapped Crossover
Over	Order Crossover

crossover and PMX are not listed in this table as they were not used in any experiments. In a problem like the TSP it would be expected that crossovers that preserve adjacency information would do better than those that work on position. For this reason, PMX and cycle crossover were not examined in these experiments because they do not preserve this information and have been shown to perform badly [62, 63, 69]. The edge-4 recombination operator was also not used as this is an expensive operator to implement and even the authors of this operator felt that the small performance improvement over edge-3 might "suggest that the point of diminishing returns has been reached" [27].

The exclusion of PMX and edge-4 leaves MPX, the other edge family operators and order crossover. Order crossover and MPX have been implemented slightly differently by different people. The term order crossover is used here to describe a version of order crossover that has two explicit crossover points, see Section 4.2.3.1 for more details. The MPX operator used took an initial segment of one third of the tour length, as has been suggested by [87] and found to perform better than a variable length segment [63].

5.2.3.5 Mutation Operators

The mutation operators used in the experiments are listed in Table 5.3. These operators were described more fully in Section 4.2.4.

Table 5.3: Mutation Operators Used in Experiments

Abbreviation	**Description**
Insert	Remove node and reinsert
Invert	Invert the path between two cities
Swap	Swap two nodes

5.2.3.6 Rank Selection

Rather than using the actual fitness values directly for selection it can be advantageous to use the ranking of the individual as an input into a function (normally linear) to calculate the expected value. This idea was discussed in Section 2.7.7.

5.2.3.7 Sample Method

A number of different sampling algorithms were used as was described in Section 2.7.2. The choices included:

- **random:** random sampling;
- **roulette:** roulette wheel sampling (stochastic with replacement);
- **universal:** stochastic universal sampling.

Random sampling was included for interest only and is not normally a GA feature since it removes the evolutionary selection pressure.

5.2.3.8 Hybrid Genetic Algorithms Techniques

A number of hybrid GA techniques were tried that made use of domain-specific knowledge to try to improve the results or the speed at which they were obtained. Normally the GA never makes use of any domain knowledge until the fitness of an individual is evaluated. With these techniques domain information is used in other places to hopefully improve the performance and efficiency of the algorithm. There may sometimes be a trade-off between these two factors because the hybrid techniques may result in problems like premature convergence [43]. Techniques used included seeding the population and using local search operators. Both of these will be discussed next.

5.2.3.8.1 Population Initialization In standard GAs, populations are normally initialized randomly. In order to improve the performance, experiments were performed in which the population was initialized using the nearest neighbour heuristic (NN) (see Section 4.3.1) and a stochastic version of the nearest neighbour (SNN) [43]. The ideas behind this were explained in Section 4.3.1.

5.2.3.8.2 Local Improvement Operators Local improvement operators were used to improve the result after genetic operations had been performed. In these experiments the only local improvement tried was a restricted version of the 2-opt heuristic as used by Lin and Kernighan (see Section 4.3.2). The restriction used was that only a single pass of improvements was made for each application in order to reduce the amount of time for the operator [12, 63]. The 2-opt heuristic normally makes repeated passes through the tour looking for 2-opt moves until a pass is unsuccessful. This restricted 2-opt is called 1-pass of 2-opt [63] by some researchers. Any future references to 2-opt should be taken to mean the restricted version described here, unless otherwise stated.

The 2-opt heuristic was used in two different places. The 2-opt could be used to improve the initial population or 2-opt could be performed on each offspring as it

was produced but before it was decided whether it would be inserted back into the population. In essence the GA was then operating on a set of local minima, as defined by the 2-opt function.

The advantage of using candidate subgraphs was discussed in Section 3.7.1.1. A nearest neighbour candidate subgraph was used to speed up performance by limiting the size of the graph to be examined during a 2-opt local improvement step. The size of the subgraph was limited by the size of the neighbourhood that was chosen. This neighbourhood size was one of the parameters that could be adjusted but was kept to a value of 10 as used in [75].

5.3 Experiments and Results

To perform the experiments, a number of GA parameters had to be selected. Also, problems had to be selected on which the GA would be run. These two choices relate to each other because, for example, some hybrid GA techniques such as 2-opt improvements can be performed only on symmetric TSPs and geometric techniques may require a metric space or perform efficiently only on a metric or euclidean space. As has been stated before, the scope of this research includes all symmetric TSPs, which in terms of implementation means that any TSP that can be represented using a symmetrical weight matrix can be handled.

5.3.1 Experimental Methodology

There are many different parameters that can be considered when designing a particular experiment. If an attempt is made to try every variation, the number of trials required and the running time quickly become explosive. However, it is still interesting to look at different combinations of parameters as there can be interaction between them.

All the experiments were run with the implemented C++ software that was described in Section 5.2 under Solaris-x86 and Linux Unix platforms on a Pentium-120 with 16M ram. All the problems used were from the TSPLIB collection of problems

[74].

The experiments were divided into the following groups:

- The aim of the first group of experiments was to decide on the sample method and the generation gap setting. It was assumed that the choices made here would be independent of the selection of other parameters. Once the best values for these parameters were chosen they were fixed and not varied throughout the second group of experiments.

- The aim of the second group of experiments was to investigate the performance and efficiency of the GA applied to the TSP. The parameters to be investigated were: mutation and crossover operators, population size, rank selection, insertion methods, handling of duplicates, population seeding and 2-opt local search

5.3.2 Group One: Generation Gap and Sample Method Experiments

The aim of the first group of experiments was to select the sample method and generation gap for the second group. The choices for sample method and generation gap have been covered by other researchers [5, 25] for other problem domains so surprising results were not expected. Only two different problems were used, shown in Table 5.4. These are both symmetric TSPs taken from TSPLIB. The optimum value and author are given as provided by [75].

5.3.2.1 Parameter Settings

All the parameter settings for this experiment are shown in Table 5.5. Three different generation gaps were tried along with three different sampling methods — stochastic with replacement (roulette wheel) and stochastic universal sampling were both tried along with random sampling where fitness information is not used at all for comparison. The population size was itself parameterized using the number of cities in the problem under investigation, represented by the variable n. Each experiment was repeated ten times.

It was expected that the choice of a smaller generation gap would result in a more opportunistic GA with faster convergence [21]. Of the three different sampling methods, stochastic universal sampling should theoretically be best, offering more consistent results on a non-steady state GA [5]. With smaller generation gap values it would be expected that this effect would be less marked. Note that no hybrid techniques, such as local improvement operators, were used in order to avoid hiding the GA-specific convergence characteristics. The crossover operator MPX was selected because it is generally found to perform well in many other studies [27, 40, 63].

Table 5.4: Problem Set Group One

Problem	**Size**	**Type**	**Optimal**	**Problem Author**
pr76	76	Euclidean	108159	Padberg and Rinaldi
lin318	318	Euclidean	42090	Lin and Kernighan

Table 5.5: Parameters Group One

Parameter	**Value**
Generation Gap	$1/p$, $0.5p$, $1p$
Population Initialization	Random
Initial Population Improvement	None
Insert Method	Average
Local Improvement	None
No Duplicates	True
Operators	MPX
Population Size	n
Problem	pr76, lin318
Rank Selection	True
Max Recombinations	20000
Sample Method	Universal, Roulette, Random

5.3.2.2 Results For Generation Gap and Sample Method Experiments

The results for the experiments are shown in Table 5.6. Each row represents the average of the ten samples that were run for each parameter set.

Table 5.6: Stage 1 Results

Problem	**Generation Gap(G)**	**Sample Method**	**Tour Length**
pr76	$1/p$	Universal	164787.7
pr76	$1/p$	Roulette	165655.9
pr76	$1/p$	Random	159963.8
pr76	$0.5p$	Universal	162813.9
pr76	$0.5p$	Roulette	166623.0
pr76	$0.5p$	Random	182056.2
pr76	p	Universal	170657.2
pr76	p	Roulette	169282.9
pr76	p	Random	218623.6
lin318	$1/p$	Universal	372939.3
lin318	$1/p$	Roulette	369832.4
lin318	$1/p$	Random	429188.0
lin318	$0.5p$	Universal	408308.0
lin318	$0.5p$	Roulette	408707.1
lin318	$0.5p$	Random	447876.2
lin318	p	Universal	433100.9
lin318	p	Roulette	434352.7
lin318	p	Random	468365.3

The effect of the generation gap has resulted in quite different convergence times, which can be seen in Table 5.6 and even more clearly in Figure 5.2 where the lin318 universal results have been plotted for different generation gaps G. The tour length has been plotted against the number of recombinations[3]. The convergence curves for

[3]Because some of these results are for generation gaps of less than one, all graphs will be plotted against recombinations and not against generations so that results can be compared.

universal and roulette sampling are plotted in Figure 5.3 for lin318 with generation gap $G = 1/p$. There is not much difference between universal and roulette methods, though the roulette method does edge out the universal method. There is, however, a larger difference between universal and roulette for problem pr76 as can be seen in Table 5.6.

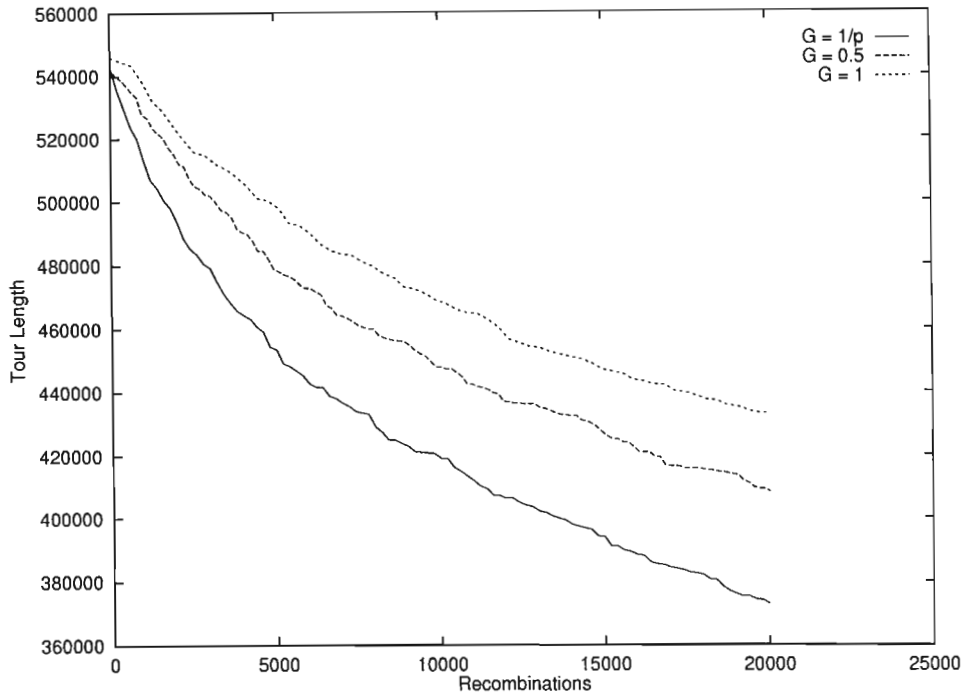


Figure 5.2: Generation Gap Comparison.

5.3.2.3 Conclusion For Generation Gap and Sample Method Experiments

The results of these first experiments show the difference in convergence characteristics with a different generation gap. In Figure 5.2 the GA with $G = 1/p$ converges substantially more quickly. The experiments proceeded to only 20000 recombinations so it cannot be conjectured how the generation gap might affect convergence over a longer experiment run where problems like premature convergence could come into effect. For this number of generations and other parameters the smaller generation gap is an advantage.

The other parameter chosen for this experiment, sample method, has not shown a clear preference for the use of either roulette or universal selection according to the results in Figure 5.3. Clearly, however, the results do show that having a selective

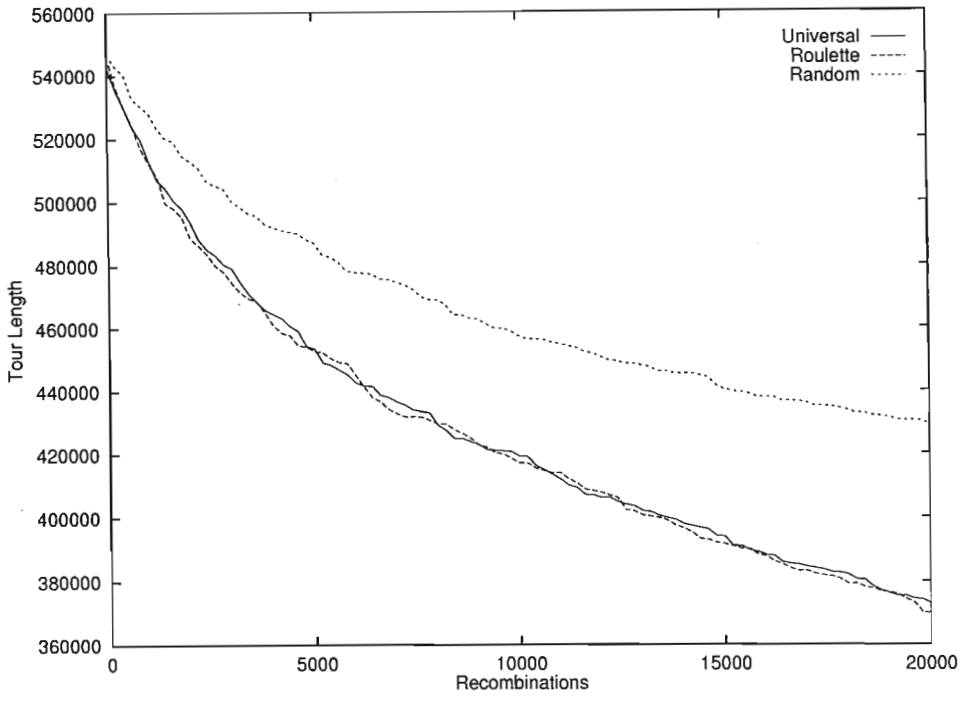


Figure 5.3: Sample Comparison.

pressure is important to the performance of this GA because of the marked difference in the final result for random selection and the other two methods.

These experiments suggest that a steady state model should be used with a generation gap of $1/p$ because of the better performance obtained from this setting. The choice between the sampling method of universal or roulette is not clear from the experiments performed here but since sampling methods become more similar as G approaches $1/p$ anyway, the choice of sampling method is not as important. The decision of sampling method can be made on other criteria like efficiency, where universal sampling is better because it requires fewer cycles through the population [5].

5.3.3 Group Two Experiments

The second group of experiments consisted of a combination of a number of parameters across a number of problems. The list of problems is given in Table 5.7. These are all symmetric TSPs taken from TSPLIB covering a number of different problem areas. The optimum values are those provided by [75]. They were selected to include a variety

Table 5.7: Problem Set Group Two

Problem	**Size**	**Type**	**Optimal**	**Problem Author**
pr76	76	Euclidean	108159	Padberg and Rinaldi
rd100	100	Euclidean	7910	Reinelt
gr120	120	Matrix	6942	Groetschel
lin318	318	Euclidean	42090	Lin and Kernighan
rd400	400	Euclidean	15281	Reinelt

of different sized problems as well as different types of problem — the problems rd100 and rd400 are randomly generated problems. The type field in the table indicates how the weights were calculated. A type of 'matrix' indicates that the weight-matrix was provided directly.

The total set of parameters that was experimented with is shown in Table 5.8. The generation gap was chosen as $1/p$ as suggested by the previous experiments. The choice of uniform sampling was based more on theory [5] then on the previous results because there was no clear indication of the superiority of either in these experiments. The motivation for the choice of other parameter values will be described in the next few subsections.

5.3.3.1 Population Sizing

It seems important to select population size correctly, as too small a population will not contain enough schemata, but too large a population will converge too slowly [42]. A number of results for population sizing exist for non-permutation GAs, in particular on binary alphabets [34]. Great population size variations have been used on the same problems. For example, a size 50 population by Eshelman [29] and a size 2000 population by Mathias [63] were both applied to the Padberg 532-city[4] with best results being 27710 and 29171 respectively (the optimum is 27686 [75]). These two GAs have significantly different structures — in particular Eshelman used a non-traditional GA, CHC, with restarts.

[4]Problem att532 in TSPLIB.

Table 5.8: Parameters Group Two

Parameter	**Value**
Generation Gap	$1/p$
Population Initialization	Random, SNN, NN
Initial Population Improvement	2-Opt, None
Insert Method	Uniform, Average, Worst, Exponentially
Local Improvement	2-Opt, None
No Duplicates	True, False
Operators	Swap, Insert, Invert, Order, Edge, Edge-2, Edge-3, MPX
Population Size	$n/2$, $n * 2$, n
Problem	pr76, rd100, gr120, lin318, rd400
Rank Selection	True, False
Max Recombinations	20000
Sample Method	Universal

A complete graph on n cities has $(n^2 - n)/2$ edges. Each tour in the population contains only n edges, so assuming each individual contains a unique set of edges, a population of size at least $\lceil (n-1)/2 \rceil$ is required for the population to contain at least one copy of each edge [63]. The construction of such a minimal population requires finding Hamiltonian cycles in incomplete graphs, which is itself an $\mathcal{NP}$-complete problem [63], so it would not be viable to construct a minimal population containing all edges. So it would seem that larger populations then $n/2$ should be used. The suggestion in [63] is that population size should be at least $O(n)$ and in some cases even $O(n^2)$. It has, however, been shown that depending on the GA implementation, it is possible to work with smaller populations. For example, if many restarts are used, it is possible to get good results with a smaller population [29]. It can also be argued that for the majority of problems many edges can be eliminated early on, as is done when using candidate sets [75].

The population sizes $n/2$, n and $2n$ were tried in this experiment. Large populations of $O(n^2)$ were not considered as they become too large for the size of problem under consideration here.

5.3.3.2 Running the Experiments

In order to test the interaction between these different parameters every combination of parameter values in Table 5.8 was tried, giving 23040 experiments, each of which was repeated five times.

5.3.3.3 Results For the Group Two Experiments

In presenting the results of the experiments, tables of percentages have been used to indicate the success of a particular parameter value. These tables were produced by looking at a table of results averaged over five runs for each problem and selecting averaged results whose tour values were within 0.5% of the top-ranked result[5]. Each parameter in turn was examined and a count was made of the number of times a

[5]A variable number were selected in this way to cater for problems like rd100 and gr120 where many good solutions were obtained for many different parameter settings. For problems such as these it would be biased to select only a fixed number of good results.

parameter value choice performed better than all other parameter value choices with all other parameters kept equal.

5.3.3.3.1 Population Size The performance of the various population sizes is shown in Table 5.9. The value for the size of the population is given as a factor of problem size. The most marked feature is the difference in performance between the large and small city problems. For the larger problems lin318 and rd400 the smaller populations have dominated the results.

Table 5.9: Population Size Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
0.5	32.0	14.5	30.9	94.4	100.0
1.0	33.8	39.1	30.4	5.6	0.0
2.0	34.2	46.4	38.7	0.0	0.0

5.3.3.3.2 Genetic Operators Table 5.10 shows the performance of the various operators. Here both crossover operators (edge, edge-2, edge-3, order and MPX) and mutation operators (insert, invert and swap) as compared. As with the population size results the importance of the operators is not very marked when dealing with easy problems like pr76 but on the larger, more difficult problems like lin318 the importance of a good operator shows up.

5.3.3.3.3 Population Initialization Table 5.11 shows the performance of different population seeding techniques. On the larger problems both the stochastic nearest neighbour and the nearest neighbour initialization have performed well. For the smaller problems the choice of this parameter does not seem to be that important for the quality of the result.

5.3.3.3.4 Initial Population Improvement Table 5.12 shows the performance of population improvement methods that are applied right after the population has been initialized. Again, for the smaller problems, there is no clear trend visible. The

Table 5.10: Operators Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
edge	2.4	0.8	0.5	0.0	0.0
edge-2	6.6	9.4	4.0	0.0	0.0
edge-3	48.3	7.1	29.1	35.7	0.0
insert	13.2	18.9	10.6	0.0	7.1
invert	14.6	28.3	13.1	0.0	7.1
mpx	5.9	7.9	18.6	35.7	78.6
order	4.2	13.4	21.1	28.6	7.1
swap	4.9	14.2	3.0	0.0	0.0

Table 5.11: Population Initialization Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
nn	27.4	30.1	19.5	50.0	58.3
random	41.2	46.8	57.8	0.0	0.0
snn	31.4	23.1	22.8	50.0	41.7

larger problems do seem to indicate a preference, but as for the population initialization, the preferences do not match.

Table 5.12: Initial Population Improvement Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
2opt	51.3	43.9	48.4	33.3	81.8
none	48.7	56.1	51.6	66.7	18.2

5.3.3.3.5 Insertion Method The performance of different insertion methods is shown in Table 5.13. Very clearly the uniform insertion method, which is just random insertion, shows up badly. This method has provided the contrast for which it was included in the experiments.

Table 5.13: Insert Method Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
average	36.1	29.3	34.9	66.7	27.3
exp	25.4	41.0	41.9	6.7	36.4
uniform	14.6	0.0	0.3	0.0	0.0
worst	23.9	29.8	22.8	26.7	36.4

5.3.3.3.6 Duplicate Handling Table 5.14 shows the performance difference when duplicates are not inserted into the population. For once there is a clear indication that use of duplicate elimination is beneficial for both small and large problems.

Table 5.14: No Duplicates Performance

Value	**pr76**	**rd100**	**gr120**	**lin318**	**rd400**
False	30.0	21.2	29.9	0.0	35.7
True	70.0	78.8	70.1	100.0	64.3

5.3.3.3.7 Rank Selection Table 5.15 illustrates the GA performance with and without rank based selection. The indicators are that rank based selection is beneficial — at least on the harder problems. The difference between the performance of rd400 and lin318 again demonstrates the different structure of these problems.

Table 5.15: Rank Selection Performance

Value	pr76	rd100	gr120	lin318	rd400
False	41.4	58.7	54.5	40.0	20.0
True	58.6	41.3	45.5	60.0	80.0

5.3.3.3.8 Local Improvement The performance of the GA with the 2-Opt local improvement operators was always better than without local improvement operators, scoring 100% across all problems. The use of the 2-Opt improvement is clearly a requirement for best performance.

5.3.3.4 Conclusion For Group Two Results

The GA is a sufficiently robust method to deal with some bad design decisions. The results given show that when a problem is relatively easy, say less than 150 cities, the selection of the parameters is not that important and even bad parameters will produce some good results. On larger problems that are consequently more difficult the selection of good parameters becomes more crucial. This is clearly seen for the rd400 and lin318 problems.

The results for population size suggest that populations of even less than $1/p$ may perform even better. They could also be viewed as suggesting that this GA is converging prematurely and so not making enough use of the extra members of the population. Both these options should be investigated with more experiments.

5.4 Comparison With Results by Other Researchers

5.4.1 A Framework for the Comparison and Own Results

The application of GAs to TSPs has resulted in a number of different approaches, perhaps because of the maturity of the TSP as a problem and the very nature of the GA which can be implemented in so many different ways. All the results reported here, with the exception of those of Chatterjee *et al* [15] make use of local search techniques to improve the results produced. Even Chatterjee *et al* has used other hybrid techniques like nearest neighbour search.

For the comparison the problems in Table 5.16 were used. The type *ATT* refers to the a pseudo-Euclidean described in TSPLIB.

Table 5.16: Comparison Problem Set

Problem	Size	Type	Optimal	Problem Author
pr76	76	Euclidean	108159	Padberg and Rinaldi
rd100	100	Euclidean	7910	Reinelt
gr120	120	Matrix	6942	Groetschel
gr202	202	Geospherical	40160	Groetschel
lin318	318	Euclidean	42090	Lin and Kernighan
rd400	400	Euclidean	15281	Reinelt
pcb442	442	Euclidean	50778	Groetschel, Juenger and Reinelt
att532	532	ATT	27686	Padberg and Rinaldi

The problems were selected based on what other researchers had studied. In particular att532 has been used as a benchmark problem by many researchers.

The results which were compared were the averages of 10 runs of the GA implementation described in this chapter. The parameters selected were based on the success of the various parameters in Section 5.3.3 and are described in Table 5.17. In addition the smaller problems (pr76, rd100, gr120, gr202) were run to 30000 recombinations with a nearest neighbour subgraph of neighbourhood 10. The larger problems

(lin318, rd400, pcb442, att532) were run to 50000 with a nearest neighbour subgraph of neighbourhood 20. These changes were made due to the results in the previous experiment which showed there is a jump in complexity between these two sizes. The operators used were both MPX and swap with a bias of selecting MPX 100 times more than swap. The results of the run are shown in the Table 5.18. The column

Table 5.17: Comparison Problem Parameters

Parameter	**Value**
Generation Gap	$1/p$
Population Initialization	SNN
Initial Population Improvement	None
Insert Method	Average
Local Improvement	2-Opt
No Duplicates	True
Operators	mpx(100),swap
Population Size	$n/3$
Problem	pr76, rd100, gr120, gr202, lin318, rd400, pcb442, att532
Rank Selection	True
Sample Method	Universal

Recomb contains the number of recombinations required to get the best results and **Time** gives the number of seconds taken at that point. The field **% Deviation** is the deviation from the optimum value. The field **Total Time** contains the number of seconds to run to completion — complete all recombinations — even if the GA converged early as was the case with pr76.

5.4.2 Review of Relevant Results by Other Researchers

Early attempts by other researchers to solve the TSP using GAs were on small problem sizes of around 10 [36] and 30 cities [47] and not all results were good. Since then it

Table 5.18: Comparative Results

Problem	**Tour Length**	**Recomb**	**Time**	**% Deviation**	**Total Time**
pr76.tsp	108159	4753.9	1.75	0.00	30.71
rd100.tsp	7966.5	6295.2	22.76	0.71	50.37
gr120.tsp	6959.5	11524	22.66	0.25	52.12
gr202.tsp	40837	9247.6	35.77	1.69	86.84
lin318.tsp	42461.2	37610.2	403.16	1.03	739.68
rd400.tsp	15474.4	39937.4	820.07	1.27	1083.44
pcb442.tsp	51529.2	44556.7	964.76	1.48	1077.39
att532.tsp	28259.5	43312.7	1150.98	2.07	1167.71

has been shown by many researchers that high quality results can be achieved using GAs and the speed even compares favourably with other methods [87].

5.4.2.1 Chatterjee *et al*

Chatterjee *et al* [15] used a GA with no crossover operation. The lack of a crossover operator is compensated for by the use of multiple mutation operators. The start population was initialized using a nearest neighbour search which was found not to adversely effect convergence but to greatly reduce the running time. No local improvement operators were used. All programs were implemented on a Vax 8600 running VMS using Pascal. Some results for Chatterjee are listed in Table 5.19. Interestingly, they claim to have improved on the result for gr666 – finding 27 and 50 improved optimal tours for population sizes of 2000 and 666 respectively. This does not seem possible as the results in TSPLIB must be proved optimal. This means there is an error in their work or in TSPLIB. They also need very few generations to produce the results for this particular problem, in contrast to the other results, suggesting a possible coding problem.

Table 5.19: Summary of Chatterjee *et al* Rresults

Problem	Population	Generations	Time	% Deviation
gr202	2000	400000	4h	2.59
pcb442	2000	320000	8h	3.5
gr666	2000	24000	12h	< 0
gr202	202	10^6	4h	3.25
pcb442	442	850000	8h	3.21
gr666	666	85000	12h	< 0

5.4.2.2 Mathias and Whitley

Mathias and Whitley [63] used the GA program GENITOR using a replace worst, linear ranking and a population of 5000. A 300000 generation run was executed with pass of 2-opt every 16000 generations. The average result for att532 was 29294 which is a 5.8% deviation from the optimum.

5.4.2.3 Ulder *et al*

Ulder *et al* set out to compare the performance of GA TSP with other methods [87]. In order to obtain the efficiency they required, they used local search. Time was fixed for each problem and different methods, including simulated annealing, threshold accepting, multiple 2-opt runs, multiple Lin-Kernighan runs, GA with 2-opt local improvement and GA with Lin-Kernighan local improvement, were tried. A summary of some of their results for the GA with 2-opt local improvement is shown in Table 5.20 and the results for the GA with Lin-Kernighan improvement is shown in Table 5.21. The labelling for the problems has been changed, in particular att532 was labelled as gro532. The time and deviations were calculated from an average of five runs on a Vax 8650 under VMS 5.1. For the 2-opt GA the populations were between 14 and 56 while the populations for the Lin-Kernighan GA were between 8 and 10. The populations were kept small in order that the times taken met the requirements of the experiment.

The results for the Lin-Kernighan GA were the best of all the results demonstrating that using the GA as a method for multiple runs would be more effective,

Table 5.20: Summary of Ulder *et al* Results For 2-Opt

Problem	Generations	Time	% Deviation
gr120	216	86	1.42
lin318	390	1600	2.02
att532	954	8600	2.99
gr666	1120	17000	3.45

Table 5.21: Summary of Ulder *et al* Results For Lin-Kernighan

Problem	Generations	Time	% Deviation
gr120	48	86	0.05
lin318	100	1600	0.13
att532	120	8600	0.17
gr666	100	17000	0.36

as information is preserved between runs. The difference between the 2-opt and Lin-Kernighan GA performance shows that it may be better to use a more expensive but more powerful local search operator even when time is restricted.

5.4.2.4 Mühlenbein and Gorges-Schleuter

Mühlenbein and Gorges-Schleuter have done a lot of work with multiprocessor GA systems and obtained some good results for the TSP att532 [40, 66, 67]. Using a 64-processor T800 Transputer network an average result over 15 runs of 27748 was obtained (a deviation of only 0.22%) [40]. The running time was three hours.

5.4.2.5 Eshelman

Eshelman has used a non-traditional GA called CHC which has the following features [29]:

- There is no bias on selection; the whole population is paired for mating.
- Parents and offspring compete to get into the next generation.

- Uniform crossover, or MPX in the case of a permutation problem like the TSP, is used rather than two-point crossover.
- Incest is avoided by not mating similar individuals.
- No mutation is performed during the recombination step — rather the whole GA is restarted after convergence using the previous population to seed the next GA run.

Using this GA he was able with a population of 50 and four restarts to obtain an average tour length of 27747 for TSP att532 (a deviation of 0.22%). This was obtained, on average, in two and a half hours on a SPARCstation 1+.

5.4.3 Comparison of Genetic Algorithm Results

An extremely long running time was required for Chatterjee's experiments, but this researcher did not make use of any local improvement operators. The GA implemented as part of this dissertation improved on the results of some researchers such as Chatterjee and Mathias. The results were also very favourable in terms of time taken but no direct comparison can be made due to the different environments and processors. The results of Ulder *et al* were improved on when compared to their 2-opt results but not when compared to their Lin-Kernighan results demonstrating again the importance of local search. The performance of Mühlenbein and Gorges-Schleuter's multiprocessor system was not beaten but this is hardly surprising given the number of processors used by them. Eshelman's CHC GA also performed better.

5.5 A Note on the Comparison of Genetic Algorithms and Other Methods for the Solution of the Travelling Salesperson Problem

This section contains a superficial comparison of the results obtained for the GA with the results of other more traditional methods for solving the TSP. When comparing

the TSP heuristic algorithms, the quality of the resulting solution is not the only important attribute of the algorithm that should be considered. One of the first reasons for selecting a heuristic approach is because perfect solutions are not possible in the time available, thus the running time of the algorithm is important. Genetic algorithms tend to be computationally expensive, so completely ignoring the speed with which a result is produced will give an unfair bias towards the benefit of the GA. For this reason both speed and accuracy have been compared.

It is recognized that some factors should also be considered when comparing heuristic algorithms including [39]:

- ease of implementation
- flexibility
- simplicity

If an algorithm is easy to implement it will be used more often and may also make more efficient use of CPU time. If a simpler algorithm produces results that are almost as good as a more complex algorithm it may well be used more often. For example, simplifications of the Lin-Kernighan have been suggested in order to produce an algorithm that is simpler and yet produces results close to those of Lin-Kernighan [61]. An algorithm is flexible if it can handle related problems. For example, an algorithm that can solve only euclidean TSPs is less flexible than an algorithm which can solve asymmetric TSPs. Simple algorithms are easier to understand and analyse and may also be easier to modify for other problem variations.

Genetic algorithms can be implemented to solve the TSP very easily and can definitely be implemented more simply than problems like the Lin-Kernighan, which is rather complex [61]. A GA environment can also be easily adapted to other problems that are similar to the TSP. The idea behind the GA is simple although a particular implementation can become complex.

Having mentioned these other factors the rest of this comparison is based on the speed and the accuracy of the GA implemented here compared with good traditional TSP heuristics.

The GA implementation described was compared against the Lin-Kernighan heuristic using results obtained by [75]. For the purposes of this comparison the results in Table 5.18 were used.

In Table 5.22 are some of the results obtained for the given test problems (shown as percentage deviations). The running time has not been shown on the table as it was not possible to obtain accurate readings for problems of this size which completed rather quickly using the optimization techniques used by Reinelt. For problems of 800 or fewer nodes the maximum running time was approximately 166s on a SPARCstation 10/20.

Table 5.22: Summary of Reinelt's Results For Lin-Kernighan

Problem	**Random**	**Nearest N.**	**Savings**	**Christofides**
lin318	1.54	2.55	2.42	0.69
pcb442	2.12	1.39	1.30	1.11

Reinelt considers a number of variants depending on the amount of back tracking done. For each variant different starting populations were considered – random, nearest neighbour, savings and Christofides. The results produced by the Christofides start were both the best achieved and the slowest.

The results in Table 5.18 are comparable with the results produced by Reinelt on some of the starts but Christofides always improves on the results developed here. On speed there is no comparison for the larger problems where the Lin-Kernighan produces results very quickly. The results by Ulder *et al* did improve on the best Lin-Kernighan results for lin318 but that is hardly surprising when it is considered that Ulder *et al*used Lin-Kernighan as a local improvement operator in their GA.

5.6 Summary

A GA was implemented and experiments run to determine good parameter values. This GA was then compared against those of other researchers with favourable results being shown but there is clearly room for improvement. In particular, a stronger local

improvement operator would seem beneficial when the results from Ulder *et al* are considered. Some of the results were also briefly compared with the Lin-Kernighan algorithm. This showed that in terms of efficiency the Lin-Kernighan is very good and hard to beat on this score.

Chapter 6

Conclusion

The aim of this research was to investigate the application of genetic algorithms to the travelling salesperson problem. This aim was fulfilled by a review of the current literature on genetic algorithms and the travelling salesperson problem, the implementation of a genetic algorithm for the solution of the travelling salesperson problem, and a comparison of these results with results obtained by other authors using the genetic algorithm and, briefly, with results obtained for the Lin-Kernighan heuristic.

The genetic algorithm was discussed in Chapter 2. Genetic algorithms have developed as part of the fast-growing research areas involving the application of natural phenomena to solve problems. The genetic algorithm is a robust, general search strategy that has the flexibility to be applied to almost any problem area. Genetic algorithms operate on a population of solutions and require very little information about the problem domain to be usable. This allows them to be applied to problems which have complex constraints. Genetic algorithms attempt to balance the exploitation of discovered information with further exploration of the search space. This makes them good at avoiding solutions that are only locally optimal which is particularly beneficial in multimodal search spaces like that of the travelling salesperson problem.

The travelling salesperson problem was examined in Chapter 3. The travelling salesperson problem is a problem of great theoretical importance which was partially responsible for the development of the field of combinatorial optimization and the theory of $\mathcal{NP}$-completeness. It can also be related to a wide range of real world appli-

cations. The study of the travelling salesperson problem is therefore important both theoretically and practically. The existence of closely related but far easier problems, such as the minimum weight spanning tree problem, make the travelling salesperson problem all the more fascinating. Although optimal solutions may be almost impossible to find for large problems, there has been considerable progress in the development of fast heuristic algorithms in the last 25 years, in particular the Lin-Kernighan algorithm.

Chapter 4 re-examined the genetic algorithm as a possible heuristic algorithm for solving the travelling salesperson problem. The first attempts at using genetic algorithms to solve the travelling salesperson problem, which were made in the early 80's, were rather discouraging. Subsequently there has been substantial progress in this area. The improvement was largely due to changes made to the basic genetic algorithm. The tours were represented using a non-binary alphabet. Rather than attempting an unnatural tour encoding so that standard 1-point crossover could be used, special crossover operators were designed that could manipulate the tour encodings without producing invalid tours. Hybrid domain-dependent features, including local optimization and population initialization, were added to improve the accuracy of the results and the speed with which they were produced.

Chapter 5 described the implementation of a genetic algorithm for solving travelling salesperson problems and the experimental results produced. The requirements of the experiments were as follows:

- to implement a genetic algorithm to solve the travelling salesperson problem;
- to examine a number of genetic algorithm parameters to determine their importance to the genetic algorithm solution of the travelling salesperson problem;
- to compare the results obtained with those of other researchers;
- to briefly compare the results obtained and those of the other researchers with the Lin-Kernighan heuristic.

A flexible genetic algorithm framework was developed in C++. This framework was designed to allow a number of different parameters to be investigated. A combina-

tion of genetic algorithm parameters was investigated, including population size, rank selection, crossover operator, mutation operator, insertion method, handling of duplicates and sampling methods. Hybrid techniques such as local search and non-random population initialization were also tried. The wide range of possibilities resulted in over 20000 experiments being performed. The use of hybrid techniques was supported by the wide use of these techniques by other researchers. Many of these different combinations were tried to avoid parameters that were only successful in a single context. Part of the reason for selecting the problems was to make it possible to compare some results with those of other researchers. The results were encouraging as the implementation produced better results than many of the results produced by them. In examining the importance of the genetic algorithm parameters, it was seen that while good solutions for easier problems can be reached with a variety of parameter settings, parameter values quickly become crucial as problem size increases. Since exponential time explosion is the major problem with the travelling salesperson problem this is an expected result. One result that stood out was that as problem size increased the smaller population sizes were more successful. This suggests that this genetic algorithm implementation was not making good use of the larger populations, either because of premature convergence or because the run was stopped before the larger population could be useful. The experiment was designed so that cases like this could be detected. It would be a good idea to experiment with even smaller populations to see if the trend continues.

The approaches taken by other researchers made changes to the traditional genetic algorithm in order to produce good results. These changes included use of distributed multiple processors, population restarts, use of a multi-cut mutation operator without any crossover, very small populations and in other cases big populations. All the researchers found it necessary to use hybrid techniques and all but one used local improvement operators of 2-opt or even Lin-Kernighan. These factors suggest that the basic genetic algorithm is not on its own well-suited to solving the travelling salesperson problem. On the other hand it demonstrates how the genetic algorithm can be used as a driver to control multiple runs of an algorithm like Lin-Kernighan when better results are required than can be obtained with a single run of Lin-Kernighan.

This is time-consuming but has the advantage of being able to be run on multiple processor systems.

The Lin-Kernighan algorithm produces very good results in a short space of time. The Lin-Kernighan implementation by Reinelt for euclidean travelling salesperson problems is particularly efficient [75, p123]. None of the genetic algorithm solutions could compete with it in terms of speed. Unfortunately there was not much overlap between problems but the percentage deviation on similar problems shows that some of the genetic algorithm results are better, although this is at the expense of being at the very least an order of magnitude slower.

There is still plenty of research that can be performed into the application of genetic algorithms to the travelling salesperson problem. This is clear from the wide variety of approaches to the problem that have been successful. It is worth investigating if some of these techniques combine well to produce even better performance. It is also possible to widen the scope by examining the application of genetic algorithms to more difficult combinatorial optimization problems, in particular, generalizations or restrictions on the travelling salesperson problem which make it unsolvable by heuristics like the Lin-Kernighan. This is where the advantages of genetic algorithms can best be demonstrated — the ability to handle complicated constraints without complicating the algorithm.

The travelling salesperson problem is an important representative of combinatorial optimization problems. The results of this research demonstrate that, with suitable modifications to the genetic algorithm's traditional form and parameters, it is possible to significantly improve its performance on such problems. However, the comparison of our results and those of other researchers with different approaches to the travelling salesperson problem indicate that there can be no complacency about the performance of genetic algorithms on such problems. In conclusion, due to the general importance of genetic algorithms for the solving of other problems the exploration of the travelling salesperson problem through this approach provides insight into these problems. The results in this dissertation and reviewed research indicate that if sufficient running time is allowed very good results can be obtained using the genetic algorithm on the travelling salesperson problem.

References

[1] ABRAMSON, D., AND ABELA, J. A parallel genetic algorithm for solving the school timetabling problem. In *Australian Computer Science Conference* (Hobart, Feb 1992).

[2] APPLEGATE, D., AND BIXBY, R. Finding cuts in the tsp (a preliminary report). Tech. Rep. 95-05, DIMACS, 1995.

[3] BÄCK, T., HOFFMEISTER, F., AND SCHWEFEL, H.-P. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 2–9.

[4] BAKER, J. E. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie Mellon University, 24-26 July 1985), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 101–111.

[5] BAKER, J. E. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 14–21.

[6] BALAS, E., AND TOTH, P. Branch and bound methods. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 10, pp. 361–401.

[7] BEASLEY, D., BULL, D. R., AND MARTIN, R. R. An overview of genetic algorithms: Part 1, fundamentals. *University Computing 15*, 2 (1993), 58–69.

[8] BEASLEY, D., BULL, D. R., AND MARTIN, R. R. An overview of genetic algorithms: Part 2, research topics. *University Computing 15*, 4 (1993), 170–181.

[9] BELEW, R. K., AND BOOKER, L. B., Eds. *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), Morgan Kaufmann, San Mateo, CA.

[10] BENNETT, K., FERRIS, M. C., AND IOANNIDIS, Y. E. A genetic algorithm for database query optimization. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 400–407.

[11] BENTLEY, J. L. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing 4* (1992), 387–411.

[12] BRAUN, H. On solving travelling salesman problems by genetic algorithms. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1* (Dortmund, Germany, 1991), H.-P. Schwefel and R. Männer, Eds., vol. 496 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 129–133.

[13] CALDWELL, C., AND JOHNSON, V. S. Tracking a criminal suspect through "face-space" with a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 416–421.

[14] CHARTRAND, G., AND LESNIAK, L. *Graphs & Digraphs.* Wadswoth & Brooks, Pacific Grove, California, 1986.

[15] CHATTERJEE, S., CARRERA, C., AND LYNCH, L. A. Genetic algorithms and traveling salesman problems. *European Journal of Operational Research 93* (1996), 490–510.

[16] CHRISTOFIDES, N. Vehicle routing. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 12, pp. 431–448.

[17] DAVIS, L. Applying adaptive algorithms to epistatic domains. In *International Joint Conference on Artificial Intelligence* (1985).

[18] DAVIS, L. Job shop scheduling with genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie Mellon University, 24-26 July 1985), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 136–140.

[19] DAVIS, L., Ed. *Genetic Algorithms and Simulated Annealing.* Pitman publishing, London, 1987.

[20] DAVIS, L. Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA.

[21] DAVIS, L., Ed. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[22] DAVIS, L., AND STEENSTRUP, M. Genetic algorithms and simulated annealing: An overview. In *Genetic Algorithms and Simulated Annealing*, L. Davis, Ed. Pitman publishing, London, 1987, ch. 1, pp. 1–11.

[23] DAWKINS, R. *The Blind Watchmaker.* Longman, 1986.

[24] DE JONG, K. A. Genetic algorithms are NOT function optimizers. In *Foundations of Genetic Algorithms 2*, L. D. Whitley, Ed. Morgan Kaufmann, 1993, pp. 5–17.

[25] De Jong, K. A., and Sarma, J. Generation gaps revisited. In *Foundations of Genetic Algorithms 2*, L. D. Whitley, Ed. Morgan Kaufmann, 1993, pp. 19–28.

[26] Dorigo, M., and Colorni, A. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics 26*, 1 (1996), 1–13.

[27] Dzubera, J., and Whitley, D. Advanced correlation analysis of operators for the traveling salesman problem. In *Parallel Problem Solving from Nature 3* (Israel, Oct. 9-14 1994), Y. Davidor, H. Schwefel, and R. Manner, Eds., Springer-Verlag, pp. 68–77.

[28] Eshelman, L., Caruana, R., and Schaffer, D. Biases in the crossover landscape. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA.

[29] Eshelman, L. J. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. Morgan Kaufmann, 1991, pp. 265–283.

[30] Fogel, D. B. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics 60*, 2 (1988), 139–144.

[31] Forrest, S., Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms* (University of Illinois at Urbana Champaign, July 17-22 1993), Morgan Kaufmann, San Mateo, CA.

[32] Garfinkel, R. S. Motivation and modeling. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 2, pp. 17–36.

[33] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[34] GOLDBERG, D. E. Sizing populations for serial and parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA, pp. 70–79.

[35] GOLDBERG, D. E., AND DEB, K. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. Morgan Kaufmann, 1991, pp. 69–93.

[36] GOLDBERG, D. E., AND LINGLE, R. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie Mellon University, 24-26 July 1985), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 154–159.

[37] GOLDBERG, D. E., AND RICHARDSON, J. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 41–49.

[38] GOLDBERG, D. E., AND SEGREST, P. Finite markov chain analysis of genetic algorithms. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 1–8.

[39] GOLDEN, B. L., AND STEWART, W. R. Empirical analysis of heuristics. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 7, pp. 207–249.

[40] GORGES-SCHLEUTER, M. Explicit parallelism of genetic algorithms through population structures. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1* (Dortmund, Germany, 1991), H.-P. Schwefel and R. Männer,

Eds., vol. 496 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 150–159.

[41] GREFENSTETTE, J. J., Ed. *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie Mellon University, 24-26 July 1985), Lawrence Erlbaum Associates, Hillsdale, New Jersey.

[42] GREFENSTETTE, J. J. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics 16*, 1 (1986), 122–128.

[43] GREFENSTETTE, J. J. Incorporating problem specific knowledge into genetic algorithms. In *Genetic Algorithms and Simulated Annealing*, L. Davis, Ed. Pitman publishing, London, 1987, ch. 4, pp. 42–60.

[44] GREFENSTETTE, J. J., Ed. *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), Lawrence Erlbaum Associates, Hillsdale, New Jersey.

[45] GREFENSTETTE, J. J., AND BAKER, J. E. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA, pp. 20–27.

[46] GREFENSTETTE, J. J., DAVIS, L., AND CERYS, D. *GENESIS and OOGA: Two Genetic Algorithms.* The Software Partnership, PO Box 991, Melrose, MA 02176, USA, 1991.

[47] GREFENSTETTE, J. J., GOPAL, R., ROSMAITA, B., AND VAN GUCHT, D. Genetic algorithms for the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie Mellon University, 24-26 July 1985), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 160–165.

[48] HEITKÖTTER, J., AND BEASLEY, D., Eds. *Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions*

(FAQ). USENET: comp.ai.genetic. Available via anonymous FTP from rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic/, 1996.

[49] HOFFMAN, A. J., AND WOLFE, P. History. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 1, pp. 1–15.

[50] HORN, J., NAFPLIOTIS, N., AND GOLDBERG, D. E. Multiobjective optimization using the niched pareto genetic algorithm. Tech. rep., Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801–2996, 1993.

[51] JANIKOW, C. Z., AND MICHALEWICZ, Z. An experimental comparison of binary and floating point representations in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 31–36.

[52] JOHNSON, D. S., AND PAPADIMITRIOU, C. H. Computational complexity. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 3, pp. 37–85.

[53] JOHNSON, D. S., AND PAPADIMITRIOU, C. H. Performance guarantees for heuristics. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 5, pp. 145–180.

[54] JONES, D. R., AND BELTRAMO, M. A. Solving partitioning problems with genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 442–449.

[55] JÜNGER, M., REINELT, G., AND THIENEL, S. Provably good solutions for the traveling salesman problem. Tech. Rep. 92.114, Angewandte Mathematik Und Informatik Universität Zu Köln, 1992.

[56] KARP, R. M., AND STEELE, J. M. Probabilistic analysis of heuristics. In *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, Eds. John Wiley & Sons, Chichester, 1985, ch. 6, pp. 181–205.

[57] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[58] LAWLER, E. L., LENSTRA, J. K., KAN, A. H. G. R., AND SHMOYS, D. B., Eds. *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization.* John Wiley & Sons, Chichester, 1985.

[59] LIEPINS, G. E., AND POTTER, W. D. A genetic algorithm approach to multiple-fault diagnosis. In *Handbook of Genetic Algorithms*, L. Davis, Ed. Van Nostrand Reinhold, New York, 1991, ch. 17, pp. 257–250.

[60] LIN, S., AND KERNIGHAN, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research 21* (1973), 498–516.

[61] MAK, K.-T., AND MORTON, A. J. A modified Lin-Kernighan traveling-salesman heuristic. *Operations Research Letters 13* (1993), 127–132.

[62] MANDERICK, B., DE WEGER, M., AND SPIESSENS, P. The genetic algorithm and the structure of the fitness landscape. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 143–157.

[63] MATHIAS, K., AND WHITLEY, D. Genetic operators, the fitness landscape and the traveling salesman problem. In *Parallel Problem Solving from Nature 2* (Amsterdam, 1992), R. Männer and B. Manderick, Eds., North–Holland, pp. 239–247.

[64] MATHIAS, K., AND WHITLEY, D. Remapping hyperspace during genetic search: Canonical delta folding. In *Foundations of Genetic Algorithms 2* (1993), L. D. Whitley, Ed., Morgan Kaufmann, pp. 167–185.

[65] MATHIAS, K. E., AND WHITLEY, L. D. Transforming the search space with gray coding. In *IEEE Conference on Evolutionary Computation* (1994), pp. 513–518.

[66] MÜHLENBEIN, H. Evolution in time and space — the parallel genetic algorithm. In *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. Morgan Kaufmann, 1991, pp. 316–337.

[67] MÜHLENBEIN, H. Parallel genetic algorithms in combinatorial optimization. In *Computer Science and Operations Research: New Developments in Their Interfaces*, O. Balci, R. Sharda, and S. A. Zenios, Eds. Pergamon Press, 1991, pp. 441–453.

[68] NAKANO, R., AND YAMADA, T. Conventional genetic algorithms for job shop problems. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 474–479.

[69] OLIVER, I. M., SMITH, D. J., AND HOLLAND, J. R. C. A study of permutation crossover operators on the travelling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 224–230.

[70] PETTEY, C. B., LEUZE, M. R., AND GREFENSTETTE, J. J. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 155–161.

[71] POWELL, D. J., AND SKOLNICK, M. M. Interdigitation: A hybrid technique for engineering design optimization employing genetic algorithms, expert systems,

and numerical optimization. In *Handbook of Genetic Algorithms*, L. Davis, Ed. Van Nostrand Reinhold, New York, 1991, ch. 20, pp. 312–331.

[72] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C, Second Edition.* Cambridge University Press, 1992.

[73] RAWLINS, G. J. E., Ed. *Foundations of Genetic Algorithms.* Morgan Kaufmann, 1991.

[74] REINELT, G., Ed. *TSPLIB 1.2.* Institut fuer Mathematik, Universitaet Augsburg, Aug. 1992.

[75] REINELT, G. *The Traveling Salesman: Computational Solutions for TSP Applications.* Springer-Verlag, 1994.

[76] RICH, E., AND KNIGHT, K. *Artificial Intelligence.* McGraw-Hill, 1991.

[77] SCHAFFER, J. D., Ed. *Proceedings of the Third International Conference on Genetic Algorithms* (1989), Morgan Kaufmann, San Mateo, CA.

[78] SCHWEFEL, H.-P., AND MÄNNER, R., Eds. *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1* (Dortmund, Germany, 1991), vol. 496 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany.

[79] SEDGEWICK, R. *Algorithms.* Addison-Wesley, 1988.

[80] SPEARS, W. M. Crossover or mutation. In *Foundations of Genetic Algorithms 2*, L. D. Whitley, Ed. Morgan Kaufmann, 1993, pp. 221–237.

[81] SPEARS, W. M., AND DE JONG, K. A. On the virtues of parameterized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 230–236.

[82] STARKWEATHER, T., MCDANIEL, S., AND MATHIAS, K. A comparison of genetic sequencing operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA.

[83] SUH, J. Y., AND VAN GUCHT, D. Incorporating heuristic information into genetic search. In *Proceedings of the Second International Conference on Genetic Algorithms* (Massachusetts Institute of Technology, Cambridge, MA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 100–107.

[84] SYSWERDA, G. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA.

[85] SYSWERDA, G. Schedule optimization using genetic algorithms. In *Handbook of Genetic Algorithms*, L. Davis, Ed. Van Nostrand Reinhold, New York, 1991, ch. 21, pp. 332–349.

[86] TAMAKI, H., KITA, H., SHIMIZU, N., MAEKAWA, K., AND NISHIKAWA, Y. A comparison study of genetic codings for the traveling salesman problem. In *Proceedings of the First IEEE Conference on Evolutionary Computation* (Orlando, FL, June 27–29 1994), pp. 1–6.

[87] ULDER, N. L. J., AARTS, E. H. L., BANDELT, H. J., VAN LAARHOVEN, P. J. M., ET AL. Genetic local search algorithms for the traveling salesman problems. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1* (Dortmund, Germany, 1991), H.-P. Schwefel and R. Männer, Eds., vol. 496 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 109–116.

[88] VOSE, M. D., AND LIEPINS, G. E. Schema disruption. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of Califor-

nia, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 237–242.

[89] WHITLEY, D., DOMINIC, S., AND DAS, R. Genetic reinforcement learning with multilayer neural networks. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (University of California, San Diego, 13–16 July 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, San Mateo, CA, pp. 562–569.

[90] WHITLEY, D., STARKWEATHER, T., AND FUQUAY, D. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms* (1989), J. D. Schaffer, Ed., Morgan Kaufmann, San Mateo, CA.

[91] WHITLEY, D., STARKWEATHER, T., AND SHANER, D. The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination. In *Handbook of Genetic Algorithms*, L. Davis, Ed. Van Nostrand Reinhold, New York, 1991, ch. 22, pp. 350–372.

[92] WHITLEY, L. D., Ed. *Foundations of Genetic Algorithms 2.* Morgan Kaufmann, 1993.

[93] WHITLEY, L. D., AND VOSE, M., Eds. *Foundations of Genetic Algorithms 3.* Morgan Kaufmann, 1995.

[94] WRIGHT, A. H. Genetic algorithms for real parameter optimization. In *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. Morgan Kaufmann, 1991, pp. 205–218.