A study of evolutionary perturbative hyper-heuristics for the nurse rostering problem

by
Christopher Stephen William Exter Rae

Submitted in fulfillment of the academic
requirements for the degree of
Master of Science in the School of Computer Science,
University of Kwazulu-Natal,
Pietermaritzburg

January 2017

As the candidate's supervisor, I have approved this dissertation for submission.

Signed: _____

Name: Prof. Nelishia Pillay

Date: _____

PREFACE

The experimental work described in this dissertation was carried out in the School of Computer Science, University of KwaZulu-Natal, Pietermaritzburg, from January 2012 to January 2017, under the supervision of Professor Nelishia Pillay.

The studies are original work by the author and have not been submitted in any form to any tertiary institution for any tertiary qualification such as degree or diploma. Where use has been made of the work of others it is duly acknowledged in the text.

_____

Professor Nelishia Pillay – Supervisor

_____

Christopher Rae – Candidate (Student number: 212561309)

DECLARATION 1 - PLAGIARISM

I, Christopher Stephen William Exter Rae (Student number: 212561309) declare that:

1. The research reported in this thesis, except where otherwise indicated, and is my original research.

2. This thesis has not been submitted for any degree or examination at any other university.

3. This thesis does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.

4. This thesis does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:
    a. Their words have been re-written but the general information attributed to them has been referenced.
    b. Where their exact words have been used, then their writing has been placed in italics and inside quotation marks, and referenced.

5. This thesis does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in thesis and in the references sections.

Signed: _____

DECLARATION 2 - PUBLICATIONS

DETAILS OF CONTRIBUTION TO PUBLICATIONS that form part and/or include research presented in this thesis:

- Publication 1:
  C, Rae, and N, Pillay. "A preliminary study into the use of an evolutionary algorithm hyper-heuristic to solve the nurse rostering problem." Nature and Biologically Inspired Computing (NaBIC), Proceedings of the 4th World Congress on Nature and Biologically Inspired Computing (NaBIC 2012). pp. 156-161. IEEE, 2012.
- Publication 2:
  C, Rae, and N, Pillay. "A Survey of Hyper-Heuristics for the Nurse Rostering Problem." Proceedings of 41st Annual Conference of the Operations Research Society of South Africa (ORRSA 2012). pp. 117-124. 2012.
- Publication 3:
  C, Rae, and N, Pillay. "Investigation into an Evolutionary Algorithm Hyper-Heuristic for the Nurse Rostering Problem." Practice and Theory in Automated Timetabling (PATAT). Proceedings of the 10th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2014). pp. 527-532, 2014.

Signed:

_____ _____

Christopher Rae Prof. Nelishia Pillay

Abstract

Hyper-heuristics are an emerging field of study for combinatorial optimization. The aim of a hyper-heuristic is to produce good results across a set of problems rather than producing the best results. There has been little investigation of hyper-heuristics for the nurse rostering problem. The majority of hyper-heuristics for the nurse rostering problem fit into a single type of hyper-heuristic, the selection perturbative hyper-heuristic. There is no work in using evolutionary algorithms employed as selection perturbative hyper-heuristics for the nurse rostering problem. There is also no work in using the generative perturbative type of hyper-heuristic for the nurse rostering problem. The first objective of this dissertation is to investigate the selection perturbative hyper-heuristic for the nurse rostering problem and the effectiveness of employing an evolutionary algorithm (SPHH). The second objective is to investigate a generative perturbative hyper-heuristic to evolve perturbation heuristics for the nurse rostering problem using genetic programming (GPHH). The third objective is to compare the performance of SPHH and GPHH.

SPHH and GPHH were evaluated using the INRC2010 benchmark data set and the results obtained were compared to available results from literature. The INRC2010 benchmark set is comprised of sprint, medium and long instance types. SPHH and GPHH produced good results for the INRC2010 benchmark data set. GPHH and SPHH were found to have different strengths and weaknesses. SPHH found better results than GPHH for the medium instances. GPHH found better results than SPHH for the long instances. SPHH produced better average results. GPHH produced results that were closer to the best known results. These results suggest future research should investigate combining SPHH and GPHH to benefit from the strengths of both perturbative hyper-heuristics.

Acknowledgements

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

I would like to thank the Centre for High Performance Computing for access to their resources and assistance.

I would like to thank my supervisor, Professor Nelishia Pillay, for her guidance and expertise in the fields of genetic programming and hyper-heuristics.

Finally I would like to thank my friends and family for their support, especially my mother and Amy Galbraith.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Equations

List of Symbols

GA	Genetic algorithm
GP	Genetic programming
HH	Hyper-heuristic
LPH	Low-level perturbative heuristic
LCH	Low-level construction heuristic
GCH	Generated Construction heuristic
GPH	Generated Perturbation heuristic
SPHH	Selection perturbative hyper-heuristic
GPHH	Generative perturbative hyper-heuristic
µA	Critical value of method A
µB	Critical value of method B
H_0	Null hypothesis
H_a	Alternate hypothesis
USP	Unwanted shift patterns
AS	Alternative skill requirement
MinCWW	Minimum consecutive working weekends
MaxCWW	Maximum consecutive working weekends
NNF	No night shift before free
D	Day off request
S	Shift off request
s1	Swap two shifts
s2	Swap a shift type with a free shift
s3	Swap n shifts
s4	Swap using problem specific conditions
s5	Swap with move acceptance
e1	Add and remove
e2	Change shift type
e3	Change n shifts
e4	Change using problem specific conditions
e5	Change with move acceptance
SE	Heuristic evolved using sprint_early5
SH	Heuristic evolved using sprint_hidden4
SL	Heuristic evolved using sprint_late6
ME	Heuristic evolved using medium_early5
MH	Heuristic evolved using medium_hidden2
ML	Heuristic evolved using medium_late3
LE	Heuristic evolved using long_early4
LH	Heuristic evolved using long_hidden5
LL	Heuristic evolved using long_late2
S	Heuristic evolved using sprint_early2, sprint_hidden1 and sprint_late2
L	Heuristic evolved using sprint_late10, medium_late4 and long_late3
H	Heuristic evolved using sprint_hidden6, medium_hidden5 and long_hidden3

E	Heuristic evolved using sprint_early9, medium_early4 and long_early4
I (I1, I2)	Iteration
A (A1–A8)	Acceptance method
C (C2, C3)	Low-level combiner
IF-C	Checks if soft constraint score has not changed
IF-I	Checks if solution has improved after executing first branch
n(n1-n13)	Low-level perturbative heuristic (LPH)
MP	Mathematical programming
ECBP	Ejection chain and branch and price
CP	Constraint programming
HHGS	Hyper-heuristic with greedy shuffle
ANS	Adaptive neighbourhood search
SVNS	Stochastic variable neighbourhood search
IP	Integer programming
HS	Harmony search algorithms
HSHH	Harmony search as a hyper-heuristic
A1	All Moves
A2	Improving only
A3	Improving or Equal
A4	Late acceptance
A5	Great deluge
A6	Step counting
A7	Simulated annealing
A8	Adaptive Iterative Limited List Acceptance
M(M1– M4)	High-level combiner

Chapter 1 Introduction

1.1 Purpose of this research

This research investigates evolutionary perturbative hyper-heuristics for the nurse rostering problem. There are two types of perturbative hyper-heuristics, selection hyper-heuristics which select heuristics that are applied to a candidate solution with the aim of improving it, and generation those that create a heuristic to affect change upon the solution.

Genetic algorithms have been employed as selection perturbative hyper-heuristics to explore the heuristic search space. These hyper-heuristics have not been applied to the nurse rostering problem. Genetic algorithm based hyper-heuristics have performed well particularly on timetabling problems and job shop scheduling. Genetic programming and variations of genetic programming have been used to create heuristics to solve combinatorial optimization problems such as vehicle routing but none have been applied to the nurse rostering problem. Genetic programming is generally used to evolve programs to solve problems; as such it is a very good fit for the needs of a generative perturbative hyper-heuristic. There are however few studies dealing with the evolving of perturbation heuristics. Through analysis of literature it was decided that a strongly typed genetic programming model would be used.

Generally selection hyper-heuristics take the form of single-point algorithms where a single mode of search is pursued. Investigating a genetic algorithm hyper-heuristic investigates a multi-point search where multiple searches are considered.

The nurse rostering problem is one of the most well known combinatorial optimization problems. It deals with the assigning of shifts to nurses in hospitals. It is also a problem that can be applied to a variety of shift scheduling problems however the problem differs from country to country and hospital to hospital. This dissertation is concerned with how effective evolutionary algorithm perturbative hyper-heuristics are when applied to solving the nurse rostering problem and the performance of the two types of perturbative hyper-heuristics.

The purpose of hyper-heuristics is not to produce the best results, but to provide a more generalized solution to problems. This is achieved through producing good results across a set of problem instances, instead of optimizing for the best results for a few instances. It is with this intent that creating two perturbative hyper-heuristic approaches is undertaken, so that these experiences can contribute to the growing body of knowledge in the field of hyper-heuristics.

1.2 Objectives

The objectives of this dissertation are:

1. Investigate a genetic algorithm selection perturbative hyper-heuristic approach for the nurse rostering problem. The approach implemented should be influenced by relevant literature.
2. Develop and analyse a genetic programming generative perturbation hyper-heuristic. This approach should create perturbation heuristics that can be used to solve the nurse rostering problem.
3. Compare the performance of the two perturbative hyper-heuristics for the nurse rostering problem.

1.3 Contributions

This dissertation contributes to the body of research on perturbative hyper-heuristic approaches for the nurse rostering problem. The following contributions are made:

- There has been little work using the genetic algorithm selection perturbative hyper-heuristics for the nurse rostering problem. This study investigates the genetic algorithm selection perturbative hyper-heuristic.
- Generative hyper-heuristics are a new field of research. Generating perturbation heuristics is newer still. This study investigates generating perturbation heuristics for the nurse rostering problem.
- A survey and analysis of nurse rostering and hyper-heuristics.
- A literature survey and analysis of the four types of hyper-heuristics is given.

1.4 Dissertation layout

The dissertation is laid out as follows:

Chapter 2 provides an introduction to genetic programming and genetic algorithms. These sections each deal with the creation of the initial population, the representation of each individual, the methods for selection of individuals, the control models and the genetic operators. The processes used by genetic programming and genetic algorithms are critically analysed.

Chapter 3 gives a summary of the most common methods used to solve combinatorial optimization problems. This is intended to provide the information necessary to have a basic understanding of mathematical and meta-heuristic approaches used for solving combinatorial optimization problems.

Chapter 4 describes the nurse rostering problem and benchmark sets that are used in the literature. Following this a state of the art literature review is given for mathematical and meta-heuristic approaches that have been used to solve the nurse rostering problem. The literature review details any relevant findings. Finally a critical analysis of the nurse rostering problem domain is given.

Chapter 5 presents an overview of hyper-heuristic approaches. It describes the four types of hyper-heuristics and reviews the literature relevant to each type of hyper-heuristic. A critical analysis of hyper-heuristics is given.

Chapter 6 presents approaches for nurse rostering using hyper-heuristics. An attempt is made to categorize low-level perturbation heuristics for the nurse rostering problem. A critical analysis of nurse rostering for hyper-heuristics is presented.

Chapter 7 outlines the methodology used to achieve the objectives of the study. First a critical analysis of the literature is given specific to creating the two approaches. The objectives of the study are restated and it is detailed how they will be achieved and measured. The details of the nurse rostering problem instances being studied are given and the parameters of the approaches are given. The details for statistical testing of the two approaches are presented. Finally the technical specifications are provided.

Chapter 8 presents the genetic algorithm selection perturbative hyper-heuristic approach. This approach uses genetic algorithm using an indirect representation to solve the nurse rostering problem. Details are given of initial population generation, the representation used, the evaluation of individuals, the selection of parents, the control model used and the genetic operators. Finally multithreading is discussed.

Chapter 9 presents the genetic programming generative perturbation hyper-heuristic approach. This chapter follows the same format as Chapter 8 providing detail on the representation used, the evaluation of the individuals, the selection of the parents, the population control model used and the genetic operators.

Chapter 10 presents the results found for each hyper-heuristic solving the nurse rostering problem. A comparison of the performance of the two hyper-heuristics is made, and a comparison to the state of the art.

Chapter 11 gives a summary of the findings and the conclusions of the investigation into perturbative hyper-heuristics for the nurse rostering problem and describes future research which will be investigated.

Chapter 2 Evolutionary Algorithms

This chapter introduces genetic programming and genetic algorithms. A brief background of evolutionary algorithms is given and then the genetic programming algorithm is presented. The terms and processes of the algorithm are introduced and discussed in section 2.2. Following this genetic algorithms are presented and the terms and processes are discussed in section 2.3.

2.1 Background of evolutionary algorithms

Theory of natural selection, population control, inheritance of traits and variation in the population take an analogy from the theory of evolution first proposed by Charles Darwin [1]. These ideas have been incorporated into various evolutionary algorithms. Evolutionary algorithm is a generic term for algorithms inspired by biological evolution. The most popular evolutionary algorithm is the genetic algorithm which was popularized by Holland [2]–[4]. The genetic algorithm started as a population of fixed length binary strings. This population was changed using the ideas of natural selection and genetic inheritance. Binary strings make up the population. Each binary string is considered an individual or chromosome. The individuals reproduce to produce new individuals in the population. Evolutionary algorithms e.g. genetic programming and genetic algorithms are used for problem solving. Evolutionary algorithms solve problems by encoding an individual to represent a possible solution to a problem.

Koza [5] introduces genetic programming where an individual in the population represents a program. Genetic programming aims to evolve a program that can solve a problem instead of a solution to a problem. It can be said that genetic programming is a search of the space of possible programs (program space) and this is fundamentally different to genetic algorithms which search a space of possible solutions (solution space) [6]. Genetic programming has been used for circuit design [7], artificial intelligence game agents [8] , data mining [9] and creation of SAT [10] and bin-packing low-level heuristics [11].

The focus of this research is on using genetic algorithms with indirect representation and genetic programming. The following sections will describe genetic programming and genetic algorithms. Genetic programming is being investigated for generating perturbative heuristics and genetic algorithms for use in a selection perturbative hyper-heuristic. Hyper-heuristics are described in Chapter 5.

2.2 Genetic Programming

This section introduces the genetic programming algorithm, depicted in Algorithm 2.1 below. Genetic programming is an evolutionary algorithm derived from genetic algorithms. Genetic programming evolves computer programs. Genetic programming iteratively refines programs using concepts taken from genetic algorithms. The processes used are selection and genetic operators.

The genetic programming algorithm presented uses the generational control model [6]. The genetic programming algorithm starts by generating a population of individuals (1). The algorithm continues until a termination requirement is met (2). The algorithm continuously performs fitness evaluation (2a) and creation of a new population (2b). The process of creating a new population ends when the new population is the same size as the existing population. Creating the new population depends on selecting parents (2bi). Genetic operators are applied to the selected parents (2bii). This generates new offspring which are inserted into a new

population (2biii). Finally the new population replaces the old population. The completion of these processes (2b) is called a generation and is a single iteration of the genetic programming algorithm. The processes are repeated until a termination criterion is met. Termination for the genetic programming algorithm can be after a set number of generations or a program is evolved that can solve the problem. The completion of the algorithm is a genetic programming run. The following sections describe the processes of the genetic programming algorithm presented in Algorithm 2.1.

1. Create an initial population randomly
2. Repeat the following tasks until a termination criterion has been reached:
    a. Evaluate the population (or Calculate population fitness)
    b. Repeat until creating a new population is complete
        i. Select parents using a selection method
        ii. Select and apply genetic operator
        iii. Insert offspring into the new population
3. Replace old population with new population
4. Repeat 2 to 3 until either
    a. End when an individual is found to be adequate
    b. Stopping criterion is met

Algorithm 2.1 The genetic programming algorithm [5]

2.2.1 Representation

Koza [5] first represented genetic programming individuals as s-expressions. S-expressions are the notation used to represent a nested list. This is the structure used to create programs in the programming language Lisp. In genetic programming an individual is a program. Individuals in genetic programming are generally represented as a parse tree [6]. A parse tree is built using primitive types called functions and terminals. Terminals in standard genetic programming are the leaf nodes. Terminals can be input variables, constants and functions with an arity of 0. Functions are branch nodes requiring arguments that can be provided by a function or a terminal. The terms used for the collection of functions and terminals in standard genetic programming are function set and terminal set [5], [12]. Searching through combinations of these primitives is called searching a program space.

Functions and terminals have an arity. The arity is the number of arguments that a primitive has. In genetic programming, functions and terminals are primitive types. Terminals all have an arity of 0 because they have no arguments. The primitives selected for use with genetic programming are usually problem specific. For example evolving a program to solve a polynomial would use arithmetic functions and evolving a program to solve a different arithmetic problem could use the same primitives. However the primitives used to solve the artificial ant problem would be unique to that problem as it uses a Boolean function IfFoodAhead that is not likely to be reused in another problem domain. An example of an arithmetic parse tree is given in Figure 2.1 where +, -, ×, ÷ represent the function set and z, x, y are inputs to the problem and hence form the terminal set. This tree would be the same as the inorder traversal: ((y×z) - (y+y)) + (x× (z-(z÷(x+y)))).

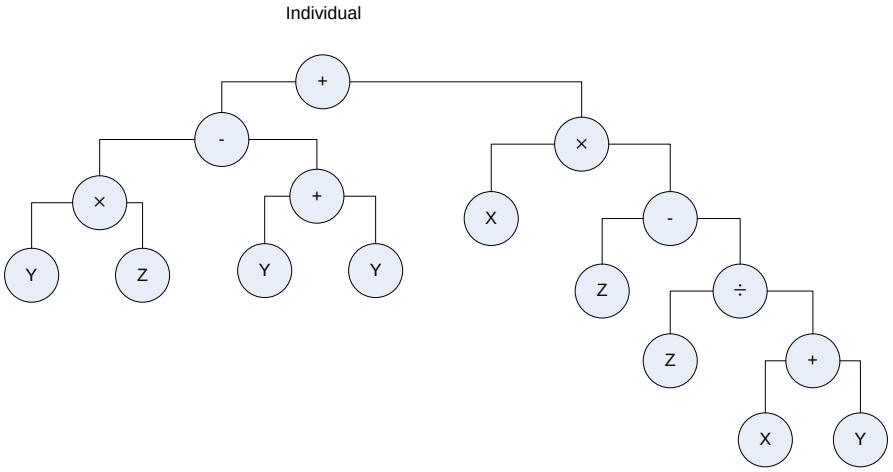

Figure 2.1 Example of a genetic programming individual

2.2.2 Initial population generation

The population is generated randomly. There are three common methods for generating a population. These methods are full, grow and ramped half and half [5]. One of the three methods is used to create the initial population.

Full creates trees that are complete at the maximum depth, these trees are not very diverse as the structure is limited. Each tree is created by selecting a random function until the maximum depth is reached. At the maximum depth a terminal is randomly selected.

Grow creates trees with variable shape. This results in trees with different depths being created. The grow method selects randomly from both the function and terminal set when creating a tree. If the maximum depth is reached a terminal is randomly selected.

Ramped half-and-half produces trees using both the full and grow methods. Half of the population is created using the full method and the other half using the grow method. Both methods create an equal number of trees for depths between 2 and the maximum depth. This is used to give a variety of tree depth.

2.2.3 Fitness Evaluation

A fitness function must be defined to measure a program's performance. The fitness function used in genetic programming depends on the problem. The fitness function is used to determine the quality of an individual. There are four common fitness functions [5]. Raw fitness is the measurement that is best suited for the problem domain, for example the food eaten by an ant in the artificial ant problem. Standardized fitness is the adjustment of the raw fitness so that a lower value is better. Adjusted fitness is an adjustment to the standardized fitness by converting the standardized fitness to a value between 0 and 1. Normalized fitness is a ratio of the individual’s adjusted fitness and the sum of the adjusted fitness of all individuals in the population. Adjusted and normalized fitness are calculated when fitness proportionate selection is used.

2.2.4 Selection

In genetic programming there are various selection methods that can be used to choose individuals for reproduction by genetic operators. The most common selection methods are tournament selection and fitness proportionate selection. These originated with genetic algorithms.

Tournament selection is the most used selection method. Tournament selection, depicted in Algorithm 2.2 is used to compare a subset of individuals with each other, returning the individual that wins the tournament. The individuals in the selected subset are randomly selected, this is called a tournament. The number of individuals in the tournament is problem dependent. A high tournament size could result in premature convergence. Tournament selection is easy to implement and results in a reduction of runtime compared to fitness proportionate selection[13].

1. Select individuals randomly from the population equal to the parameter tournament size
2. Set the best individual as the first individual in the sample
3. Repeat until there are no individuals to compare against:
    a. Compare the best individual's fitness with the next individual in the sample
    b. If the fitness of the best individual is better than the compared individual, replace the best individual with the compared individual.
4. Return the best individual

Algorithm 2.2 Tournament selection [14]

Fitness proportionate selection requires the calculation of the normalized fitness of each individual in the population to assign a fitness value proportionate to the entire population. This is also referred to as roulette-wheel selection [15]. A ‘mating’ pool is created to store the number of occurrences of each individual. Individuals with poor normalized fitness will virtually never be selected. This can reduce the diversity of the population and result in premature convergence. Due to the calculations involved this results in fitness proportionate selection being more computationally expensive compared to tournament selection [5].

1. Calculate the standard fitness
2. Calculate the adjusted fitness
3. Calculate the normalized fitness
4. Create a ‘mating pool’ by performing the following for each individual:
    a. Multiply the normalized fitness by the population size
    b. Round this value and assign it to the individual
    c. Insert into the ‘mating pool’
5. Randomly select an index from the ‘mating pool’
6. Return the individual of the index selected

Algorithm 2.3 Fitness proportionate selection [16]

2.2.5 Genetic operators

Genetic operators are used to create new offspring. Offspring are individuals that have been created using a parent individual and applying a genetic operator to the parent. Some genetic operators promote convergence and some diversity. It is necessary to find a balance of convergence and diversity for the evolution process to succeed.

2.2.5.1 Crossover

Crossover is a local search operator because it searches a subarea of the program search space. The operator randomly selects two points, one in each of the selected parents. The subtrees at these two points are swapped. This generates two new offspring. In Figure 2.2, an example of this can be seen where the subtree rooted at the arrow pointing to the minus function in Parent 1 is exchanged with subtree in Parent 2 shown with an arrow. This results in two new offspring; offspring 1 and offspring 2 [12].

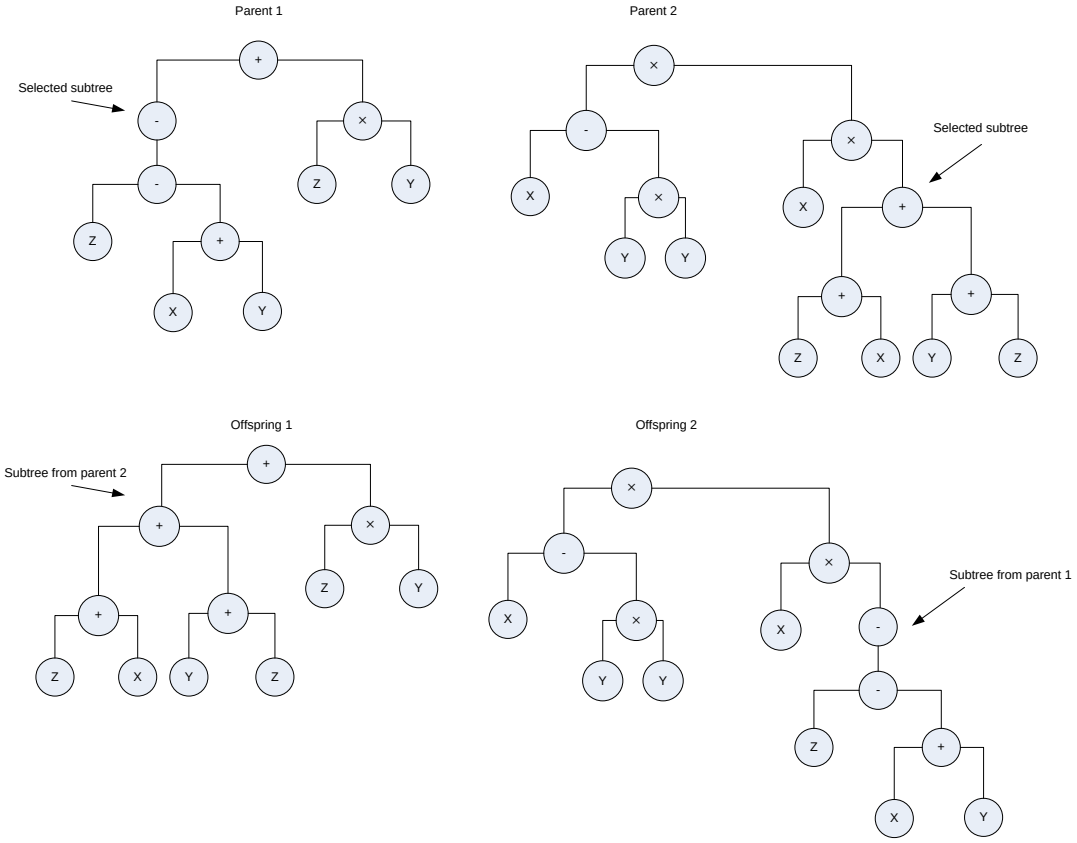

Figure 2.2 Standard genetic programming crossover

2.2.5.2 Mutation

Mutation is a global search operator. In tree based genetic programming subtree mutation is common and has a number of variants. Standard subtree mutation selects a random point in an individual and creates a new subtree at that point using the grow method [12]. Initially mutation was not used for genetic programming by Koza as he wanted to prove that genetic programming was not a random search of the program space [6]. However it has since gained popularity. Koza suggests very little mutation be used [17]. An example of subtree mutation can be seen in Figure 2.3 where the arrow pointing to the multiplication function in the second branch is selected and this entire subtree is replaced with a new subtree.

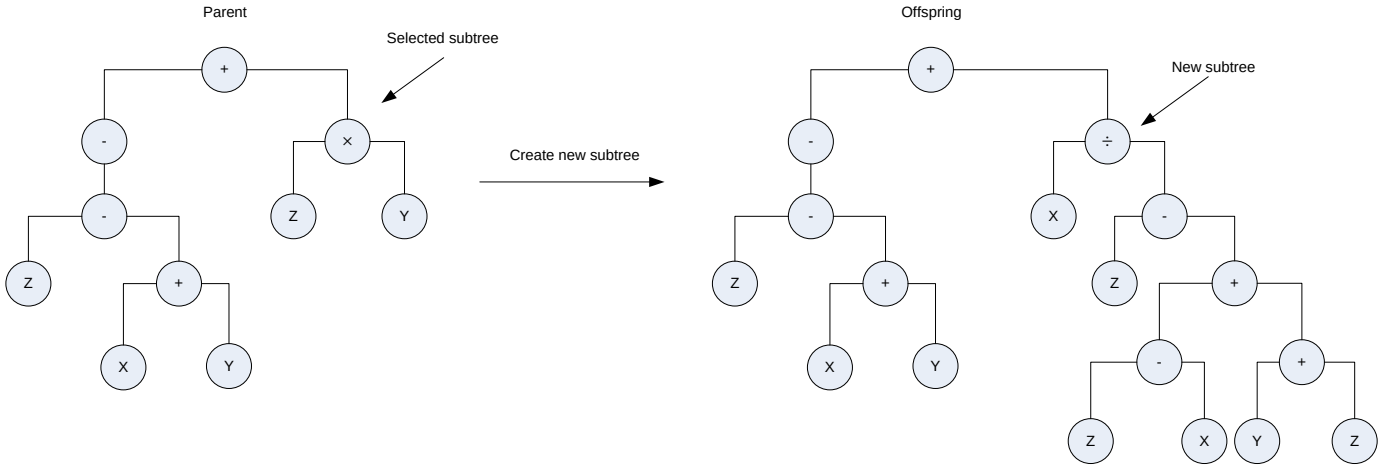

Figure 2.3 Subtree mutation operator

Point mutation [12] is an example of a mutation operator that does not change the structure of a selected subtree. Point mutation is where single node in an individual is selected and changed to a different compatible function or terminal[6]. For example changing a function + to a × or a terminal z to x would be valid point mutation.

2.2.5.3 Create operator

The create operator is a global search operator that generates a new individual. The create operator introduces new individuals into the population using the method used for the creation of the initial population. This would be either the grow or full methods [18], [19].

2.2.5.4 Permutation

Permutation is a local search operator but it is still considered a type of mutation [12]. Permutation results in a new arrangement of an individual. Permutation used by Koza[5] selects a random node in a tree and swaps the arguments of the selected node to create a new permutation of those arguments as a new individual. This operator has also been referred to as a swap operator [6] and is similar to the inversion operator which swaps subtrees that are not contained within each other. An example of permutation can be seen in Figure 2.4 where the plus function on the extreme right hand side of the tree is exchanged with the terminal Z within the subtree. This is a simple example of permutation.

2.2.6 Control models

Control models define how a population is evolved [6], [20]. In genetic programming the most common control model is the generational control model. A genetic programming algorithm using the generational control model uses a population of fixed size which is iteratively refined until a set generation limit is reached [19]. This is illustrated in Algorithm 2.1. Each iteration is called a generation. For each generation, the entire population is replaced with new individuals created using genetic operators. The offspring created for each generation is determined by genetic operator application rates set for the run. An application rate is a proportion of the population. For example a 0.9 crossover rate with a population of 100 individuals would result in 90 individuals of the next generation being created using crossover.

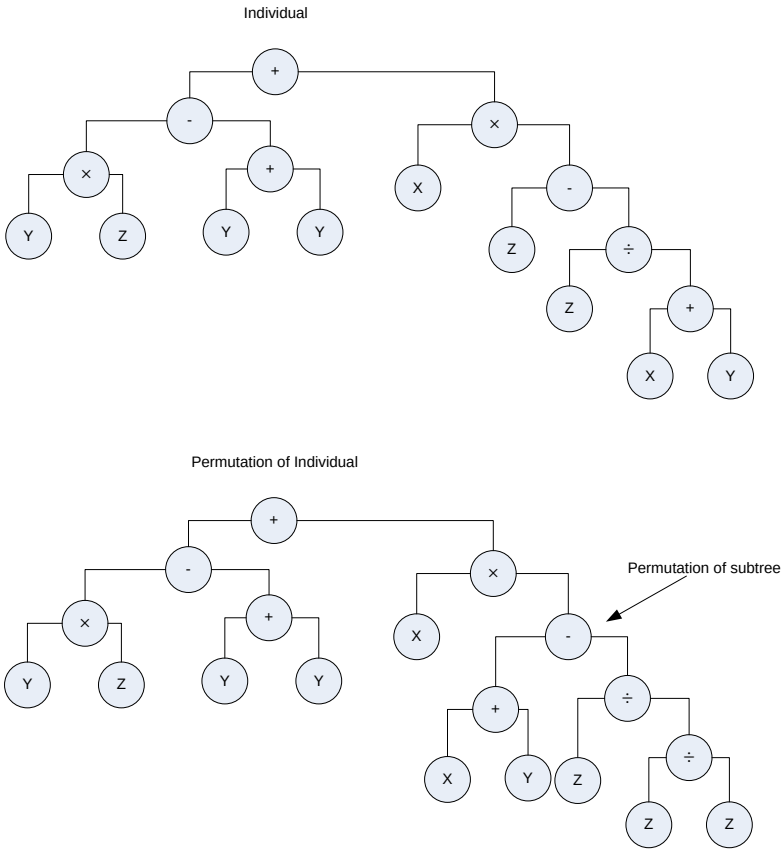

Figure 2.4 Permutation operator

The steady state model differs from the generational model, in that a single population is maintained [21]. Individuals are still selected using tournament selection but genetic operators now replace weaker individuals using inverse tournament selection. This is done instead of replacing the entire population. The number of offspring replaced at each generation is generally very small. The steady state control model used in [18] achieved almost twice the number of successful runs for evolving a sorting algorithm than the generational control model.

2.2.7 Strongly typed genetic programming

Strongly typed genetic programming enforces type constraints [22]. This ensures the evolution of legal programs by the genetic programming algorithm. This is done by defining which argument types are acceptable for a function and the return type of the function. An example is a function called 'dot product' that takes 2 arguments of type 'vector' and returns a scalar number. Genetic operators must be implemented so that only valid offspring are produced. Valid offspring are trees where all functions have children of the correct type. In the initial population only legal trees are created. For mutation this means using the same method of creation as the initial population to create syntactically correct individuals. For crossover the change means that only nodes of subtrees which are the same type may be exchanged between individuals.

2.2.8 Critical analysis of genetic programming

Initial population generation is important as it affects diversity which in turn affects premature convergence. It is ideal to avoid prematurely converging to an area of the search space, where a

suboptimal solution is found. In this chapter three methods for generating the initial population were described, namely full, grow and ramped half-and-half. The full method limits tree depth but could result in premature convergence as a result of a lack of diversity in the population because the trees are all of the same depth. The grow method encourages a diverse population of varying depths. Ramped half and half is the best of both worlds ensuring that some trees are created using the full method and some using the grow method. Grow will still give populations with good diversity.

The generational control model is generally employed in genetic programming. The model replaces the old population with a new population under the assumption that the new population is better than the previous population. The steady-state model is considered less likely to suffer premature convergence [23]. It has been found in genetic algorithms that the steady-state model reduces the number of generations needed to find a solution [24]. In the steady-state control model a few of the weaker individuals are constantly removed via an inverse selection method. The steady-state model has been found to be effective in genetic algorithms and grammatical evolution [25].

The selection method used to choose the parents for the genetic operators could result in premature convergence by being too elitist. Generally tournament selection is used instead of fitness proportionate selection in genetic programming. Tournament selection is commonly used because it is easy to implement and generally has better runtimes than fitness proportionate selection [13]. Fitness proportionate selection is more likely to encourage premature convergence as solutions that are better have a higher chance of being selected. Tournament selection firstly selects a random sample of the population. The size of this sample is determined by the tournament size. The tournament size is used to apply selection pressure. A large tournament size results in elitist selection and potentially causes premature convergence. A small tournament size would exert very low selection pressure especially if the population size is high. It would also increase the time it takes for the GP algorithm to converge. It seems that generally a tournament size of 5 is the most commonly used value, it still allows for a decent subset of the population to be selected. This means that poor quality solutions may not be entirely excluded due to lower selection pressure.

Population size is an important factor in maintaining diversity, if a population is too small diversity will not exist and premature convergence will occur. This can be because the small population might not fully represent the program space. If a population size is too big, convergence may be slow. The initial populations in genetic programming are usually created without duplicates to increase the diversity of the population.

Genetic operator application rates usually include more crossover than mutation as Koza [5] initially wanted to demonstrate that genetic programming is not a random search. There has been work that has only used mutation operators[26]. Luke and Spector [27] found that crossover is generally more successful than subtree mutation but subtree mutation works well with small population sizes. The results were not statistically significant when comparing the two operators. Mutation does increase the diversity of the population. Crossover is necessary for the genetic programming algorithm to converge. Ideally the goal of genetic programming is to generate a program that is capable of solving the problem at hand. If the genetic programming algorithm is unable to converge to an area of the search space, it may require extensive runtimes. Mutation is necessary to increase diversity in the population and prevents premature convergence. With regards to other operators, there is little work done on their effectiveness and applicability. It can be inferred, however that global search operators such as create will perform a similar role to mutation while local search operators like permutation will perform a similar role to crossover. Global search operators increase diversity in the population and local search operators promote convergence.

The aim of genetic programming is to evolve a program that can solve a problem. For some problem domains the aim is to produce a solution program with an expected result that can be identified. There are problem domains, for example the field of combinatorial optimization problems, where the global optimum is unknown and the aim is to minimize or maximize the cost of the solution produced by the evolved program. A good fitness function can create a distinction between poor programs and good programs. It is however, difficult to tell whether a fitness function results in poor or good performance of evolved programs. In general a genetic programming algorithm can end execution when an adequate solution is found but in the cases where an adequate solution is unknown or the goal unspecified other termination criteria must be considered. The most common termination criterion used in these instances is setting a limit on the number of generations. This is effective as it provides a clear termination criterion for the execution of the genetic programming algorithm. Another method may be to enforce some type of convergence checking where if the individual with the best fitness has not changed for a number of generations then the run could be ended. This indicates the genetic programming algorithm has converged. Due to the stochastic nature of evolutionary algorithms an ideal solution is not guaranteed with every run. As such it is advisable to abandon runs that have converged to a poor area of the search space. Generally the best approach seems to be placing a limit on the number of generations to be performed in a run. In most genetic programming studies a generation limit of 50 generations is usually used and this seems to be sufficient for convergence.

The primitives used dictate the structure of the individuals in the population and therefore the diversity of the population. A terminal set and or function set that do not sufficiently represent the problem may not be able to produce a solution. A large primitive set (terminal set and or function set) would increase the search space. The function set can consist of arithmetic, Boolean, problem specific functions or combinations of any of the above. For example in the artificial ant problem it is common to use the problem specific function IfFoodAhead but this function would not be useful in other problem domains. It is easy to choose a function set when a problem has a set type, such as arithmetic problems, as then one only need consider arithmetic functions. There are however, many mathematical functions that could be chosen e.g. sin, cos, modulus etc. in addition to simple arithmetic operations e.g. plus, minus, division and multiplication. It is generally unclear on how one decides on a function set and what affect the chosen functions have on the genetic programming algorithm. Obviously not including a diverse range of functions can result in premature convergence. This is because a solution may be impossible to find. It is assumed that evolution would penalize primitives that do not assist in finding the desired solution or may impact the evolved program's performance. Evolution does not strictly eliminate primitives that hinder the search process from being a possible option for an offspring created in later generations. It is possible to eliminate primitives completely by using no global search operator e.g. mutation. A biased fitness function can also eliminate primitives. A good fitness function can help the evolutionary algorithm avoid individuals that consist of poorly performing combinations of the function and terminal sets. A large function set does increase the search space but too small a search space may result in programs that do not result in an adequate solution. It seems best to include functions that are specific to the problem domain. This can be determined through knowledge of the problem domain.

Genetic programming is a powerful method for evolving programs. It is capable of exploiting the advantages of the population based approaches by exploring a large search space and considering multiple solutions. The major disadvantage is that potentially not all runs will be successful. The stochastic nature of the genetic programming algorithm can lead the search inadvertently to premature convergence. The primitives may not be able to construct a program solution. Changing parameters such as genetic operator application rates can result in an increased chance of finding a solution. It is difficult to find the correct set of parameters. Genetic programming results in a program, this has potential to be induced quicker than an

algorithm designed by a human programmer. An evolved program can be more reusable than a human designed algorithm. Genetic programming has the power of automation where it can allow the evolution of new programs for problem domains that may be difficult for a human programmer to create. It has already been shown that evolving programs is known to have high runtimes but has been found to be competitive with human programmers coming up with solutions to solving problems. There still exists a large amount of research to be done in genetic programming such as identifying methods for choosing ideal parameters, function and terminal sets and whether the current evolutionary control models and genetic operators are the best that are available. Currently many of the choices made in applying genetic programming are best guesses and even slight changes in genetic operator application rates result in solutions being found or not. It is difficult to choose parameters for genetic programming, this is because each parameter change may affect another parameter. Genetic programming runs are generally quite expensive computationally and having to run a number of tests to determine if a parameter is impacting the results is a further time cost. Parameter tuning tools are starting to be used in the field of evolutionary algorithms. There is little research into the effectiveness of using parameter tuning tools for genetic programming. Runtimes are also an issue for genetic programming when large data or complex evaluations are necessary. This will be alleviated through using distributed computing using multi-core architecture.

2.3 Genetic algorithms

Genetic algorithms were first implemented as a method to use evolutionary ideas to optimize a population of fixed binary strings [3]. A binary string is made up of a number of bits, each bit is treated as a part of the solution but requires decoding to be relevant. Each bit can represent a different feature of the problem. This encoding is not ideal for all problems. For example a travelling salesman problem chromosome would be made up of n bits to represent the n cities which must be visited. A single bit changing can create an illegal solution as it is not legal to visit the same city twice. An alternate to binary representations are direct representations of the solution space for solving problems [28], [29]. Direct representations can be more appropriate for example a representation for the travelling salesman problem would be character string. Each character in this character string would represent a city. An indirect representation would differ from both a binary encoding and direct encoding by instead of representing a solution space it would represent operators which should be applied to a solution space. Studies found success using genetic algorithms with an indirect representation where the problem was solved by the decoding of the chromosome string [30]–[32]. Selection perturbative hyper-heuristics employing a genetic algorithm to explore the low-level heuristic space are essentially genetic algorithms using an indirect representation [33], [34]. This section provides an overview of genetic algorithms using an indirect representation.

2.3.1 The genetic algorithm

The genetic algorithm presented by Goldberg [15] is depicted in Algorithm 2.4. This is considered the first major use of an evolutionary algorithm for problem solving.

The algorithm begins by creating an initial population and evaluating each individual (1). Then the algorithm enters a loop (2) which will continue until a new population of offspring is created. In order to generate individuals for the new population, fitness proportionate selection is used to select two individuals as parents (2a). Genetic operators are applied to the selected parents (2b). Two offspring are produced by process 2b. If a random number is in the range of the probability of crossover, crossover is applied to the selected parents. If this random number is not in this range then the selected parents are copied into the next generation (2bi). Following crossover if a random number is within the probability for mutation, mutation is applied to both

offspring (2bii). Different probabilities are used for crossover and mutation. For example if the crossover probability is 80% (0.8) and a random number generated in the range of 0 and 0.8 is generated then crossover would be applied to the selected parents. Probability outside these bounds will result in crossover not being applied. The offspring are then evaluated (2biii) and inserted into a new population (2biv). Once there are the same number of individuals in the new population as the original population, the new population replaces the old population (3). This is considered a generation as part of the generational control model described in section 2.2.6. The steady-state control model described in section 2.2.6 is also used for genetic algorithms. In order to change the above algorithm to a steady-state model one would instead insert the created offspring into the current population using an inverse selection method to replace individuals with poor fitness in the population.

1. Create an initial population and evaluate
2. Repeat the following tasks until a new population is created:
    a. Select two individauls from the population **i1** and **i2**
    b. Repeat until creating a new population is complete
        i. **If** crossover probability
            Perform crossover to **i1** and **i2** → creating offspring **i3** and **i4**
            Else
                Copy i1 and i2 → i3 and i4
        ii. **If** mutation probability
            Perform mutation to **i3** and **i4**
        iii. Evaluate **i3** and **i4.**
        iv. Insert **i3** and **i4** into new population
3. Replace old population with new population
4. Repeat 2 to 3 until termination criterion is met

Algorithm 2.4 The genetic algorithm [15]

2.3.2 Similarities to genetic programming

Genetic programming covered in section 2.2, shares common features with genetic algorithms. Fitness functions and selection methods are generally the same used between the two evolutionary algorithms. Fitness functions are the same as those used by genetic programming (section 2.2.3). Genetic algorithms used to employ fitness proportionate selection [35] for selection. Tournament selection as described in section 2.2.4 is also used for genetic algorithms. Tournament selection gained popularity in genetic algorithms after the introduction of genetic programming.

2.3.3 Initial population generation and representation

The initial population is generated randomly, each character in the string is randomly chosen. Each individual in a genetic algorithm is called a chromosome. Each chromosome in the population is encoded using a binary string. A binary chromosome is decoded and maps on to a solution space, a direct genetic algorithm maps directly to a solution and a genetic algorithm using an indirect representation requires decoding to create a solution [36]. A binary string chromosome could be "10011010" which would be two bits and may represent two digits necessary for a solution. A direct genetic algorithm would map the solution space directly to

the individual. For example, Raghavjee [30] used a two-dimensional matrix to represent a school timetable. Unlike the standard genetic algorithm which uses a binary string representation a genetic algorithm using an indirect representation maps an integer or character string to operators which are applied to a candidate solution. For example the string "13326432" would be an example of this. Each gene (integer) represents a different operator to use to change a candidate solution. Han et al. [31] use an indirect representation of integer strings where heuristics are mapped to each integer in the string. The genetic algorithm using an indirect representation by Han et al. [31] was shown to outperform a direct genetic algorithm for the trainer scheduling problem. Corne and Ogden [32] used a genetic algorithm using an indirect representation which was better than a direct genetic algorithm at solving the preacher timetabling problem. Raghavjee [30] uses a character string representation for a genetic algorithm using an indirect representation for solving the school timetabling problem. Raghavjee [30] found the genetic algorithm using an indirect representation to be better at solving the school timetabling problem than the genetic algorithm using a direct representation. The representation used should be appropriate to the problem domain [37] as mentioned earlier a binary encoding would not make sense for a travelling salesman problem in which each city only needs to be visited once.

2.3.4 Variable length chromosomes

Standard genetic algorithms used fixed length individuals but variable length individuals have been used with indirect representations. The messy genetic algorithm is considered the first evolutionary algorithm to feature variable length individuals [35], [38]. This representation allows for complex representations such as that needed for the prisoner's dilemma problem as researched by Lindgren [36], [39]. Messy genetic algorithms use a cut-and-splice crossover operator, resulting in variable length individuals. Han et al. [31] show an improvement in using a variable length representation to their previous fixed length genetic algorithm using an indirect representation [31].

2.3.5 Genetic operators

Genetic operators are used to evolve and create new individuals for each generation with the intent of improving the fitness of each individual. The most common operators used for genetic algorithms are crossover and mutation.

2.3.5.1 Crossover

For variable length individuals the crossover operator generally used is the cut and splice operator introduced by Goldberg [35]. In Figure 2.5 the cut and splice crossover is presented where a point is selected in each parent. In parent 1 the point selected is position 4 and the operator cuts the rest of this string. In parent 2, position 5 is selected. The first offspring, offspring 1 is the combination of parent 2 (4-3-2-5-7) with parent 1 of (2-5-7-6-3-3). The second offspring, offspring 2 is the combination of parent 1 (1-4-3) and parent 2 (6-3-1-2-2-4). The offspring are created by taking segments from the selected points. Offspring 1 is given from positions 1 to 5 of parent 2 and positions 4 to 9 in parent 1. Offspring 2 is created from positions 1 to 3 of parent 1 and position 6 to 11 of parent 2.

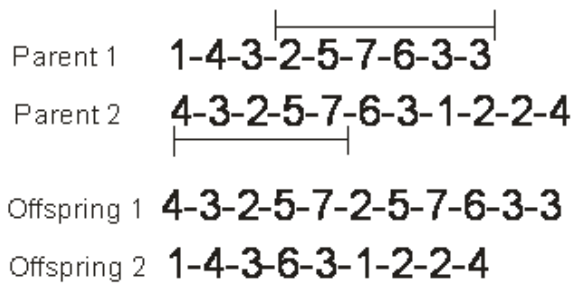

Figure 2.5 Cut and splice crossover

2.3.5.2 Mutation

The mutation operator randomly changes a character in an individual. It replaces a selected character with a randomly selected character. In Figure 2.6 an individual is mutated with the character at position 5 in the string changing from 5 to 3 [36].

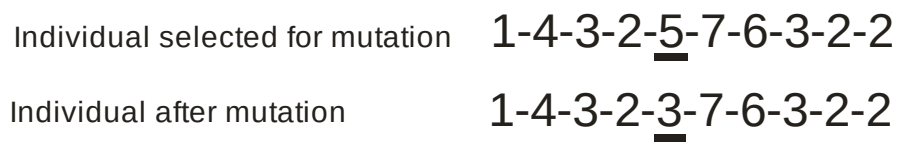

Figure 2.6 Mutation

2.3.6 Critical analysis of genetic algorithms

The genetic algorithm is a flexible meta-heuristic approach that is capable of solving problems by exploring the solution space as a multi-point search. In this way a genetic algorithm is often able to overcome the limitations of single-point searches such as tabu search, hill climbing and variable neighbourhood search, by not being easily stuck in local optima. Representation plays a key role in how effective a genetic algorithm is for the problem domain it is applied to. Firstly the representation choice is between a direct encoding against an indirect encoding. Genetic algorithms for timetabling and scheduling problems have generally performed better using an indirect representation. It is generally the case that a variable length is better when using an indirect representation [31], [34]. Using variable length individuals has been shown to be effective in various domains using genetic algorithms as well. As such a genetic algorithm using both a variable length and indirect representation generally uses cut and splice crossover. In terms of selection pressure tournament selection is more efficient than roulette wheel selection and been shown to be effective as a selection method for genetic algorithms. The generational model is the most commonly used model used for genetic algorithms. The representation best suited to scheduling and timetabling problems is an indirect representation mapping characters to heuristics [32], [34], [40].

2.4 Summary

This chapter presents an overview of evolutionary algorithms. Genetic programming is presented, the algorithm, the representation, initial population generation, selection methods, genetic operators and control models are detailed. A critical analysis of genetic programming is then given. Genetic algorithms are then presented, the algorithm, the similarities to genetic programming, initial population generation and genetic operators. Finally a critical analysis of genetic algorithms is given.

Chapter 3 Methods used for solving combinatorial optimization problem

This chapter provides a summary of methods used by studies mentioned in Chapter 4, which reports on the state of the art survey of methods applied to nurse rostering and Chapter 5, a survey of hyper-heuristics. The majority of these methods deal with solving combinatorial optimization problems for example, bin-packing, the travelling salesman problem, timetabling and nurse rostering. Section 3.1 and section 3.2 provide an overview of mathematical methods and meta-heuristics respectively.

3.1 Mathematical methods for combinatorial optimization

This section provides an overview of mathematical methods that are used to solve combinatorial optimization problems. The first two methods covered are integer linear programming (Section 3.1.1) and constraint programming (Section 3.1.2). These are methods that are used to solve combinatorial optimization problems by creating mathematical models. The final section looks at branch and bound, a popular exact method for solving combinatorial optimization problems (Section 3.1.3).

3.1.1 Integer linear programming

Integer linear programming (ILP) models optimization problem constraints using integer variable representations [41]–[43]. Integer variables are used to represent the constraints of the problem. ILP supports both discrete and continuous variables. Two popular variants of ILP exist: mixed integer linear programming, where variables are represented using real numbers, and 0-1 linear programming which represents all variables using binary values. Once a problem has been formulated as an ILP problem it can be solved using either an exact method such as branch and bound [44] and or a meta-heuristic method [45].

3.1.2 Constraint programming

Constraint programming is where a problem is modeled in terms of a set of problem specific constraints and variables [43]. Constraint programming allows for specialized constraints modeling. It is limited to modeling discrete problems and has no limitation on arithmetic constraints for decision variables. Each variable represents a single value that must be set from a predetermined range. An example variable for the nurse rostering problem (see Chapter 4 for details) is Nurse-Day, where each nurse would have a Nurse-Day for each day that needs to be scheduled. This variable holds the value for a shift e.g. E - early shift, N - night shift or L - late shift [46]. The variables used depend on how the problem is modeled. The variables can be represented by integers, strings or a problem specific type. The aim is to then minimize the constraints. The minimization of these constraints is achieved by using an exact method e.g. branch and bound [47] or backtracking [48], and or meta-heuristics [49]. The values of variables are restricted by the constraints. Constraint satisfaction is the most common type of constraint programming used for complex problem domains [50].

3.1.3 Branch and bound

Branch and bound and known variants; branch and price and branch and cut, are considered exact algorithms [51]–[53]. The branch and bound algorithm uses a lower bound calculation method and an upper bound calculation method to attempt to find the optimal value. The lower bound calculation is either a heuristic or a method e.g. Lipschitz [54]. The upper bound calculation is either a heuristic or an optimization method such as the simplex method which will set the best solution found so far as the upper bound. Branch and bound first calculates the upper and lower bounds of an initial candidate solution created randomly or using a low-level construction heuristic (LCH, see Chapter 5 section 5.3). Branch and bound then creates branches using a partitioning algorithm, specific to the problem being solved, to partition the solution into two new possible candidate solutions. Each branch is a slightly different variation of the initial solution and each created branch has its upper and lower bound calculated. A branch that results in a feasible solution and a solution with equal or improving objective value is selected and branched. This process of branching and updating the upper and lower bounds, is repeated until an optimal solution is found. Branch and price is a variant which uses column generation to update a master problem (the original problem), the restricted master problem (only considers a subset of columns of the master problem) and the pricing problem, which is a sub problem that is solved to find a column with a negative reduced cost. The pricing problem is solved with a heuristic or local search method. Columns with a negative reduced cost are added to the restricted master problem [55]–[58]. Branch and cut uses a cutting plane algorithm, which iteratively refines a candidate solution by generating and solving linear inequalities. This cutting is only done when it is valid to do so after the branching of the problem [59], [60].

3.2 Meta-heuristics

Meta-heuristics explore and exploit the search space for the particular problem domain. The aim of meta-heuristics is to find an optimal solution to a problem in a reasonable time period. Where the global optimum is unknown, finding a solution as better than the current best known solution is the goal [61], [62].

Meta-heuristics that will be described in this section are all local search methods. Local search iteratively improves an initial solution. This is achieved by searching a neighbourhood of solutions. A neighbourhood is a search space of a candidate solution where small changes to the candidate solution are considered neighbouring because they are similar to the original solution. This is done by applying one or more neighbourhood operators to a candidate solution. 2-opt is an example of a neighbourhood operator [63]. The operator was first developed to solve the travelling salesman problem. 2-opt can be used on any problem that can be mapped to a graph. The algorithm swaps the order of two points of a graph. Neighbourhood operators result in small changes to a candidate solution, producing a neighbouring candidate solution. Applying a neighbourhood operator is moving from a solution **S** to a solution **S'**. A local search is performed usually for a set number of iterations. Each iteration compares the difference between **S** and **S'** until a local or global optimum has been identified [64].

3.2.1 Hill climbing

Hill climbing is a local search algorithm [65]. The hill climbing algorithm is given in Algorithm 3.1 the algorithm starts with an initial candidate solution **S** (1). This initial solution is generated through LCHs, heuristics that build an initial solution using a rule of thumb such as put the smallest container in the box first, or randomly initialized. For a number of iterations (**kMax**) the algorithm attempts to find an improved solution. This is done by applying a neighbourhood operator resulting in a new solution **S'** (**2a**). This new solution **S'** is compared to **S**, if **S'** is an

improvement then **S** is set to **S'** (**2c**). **A** counter variable **k**, with a starting value of 0 is incremented (**2c**) and once the iteration limit is reached a solution is returned.

1. Start with an initial candidate solution ***S***
2. Repeat the following tasks while **k** < **kMax**:
    a. Apply neighbourhood operator resulting in neighbouring solution **S'**
    b. If **S'** is an improvement on **S**
        S = S'
    c. k++
3. Return best found solution

Algorithm 3.1 Hill climbing [65]

3.2.2 Tabu search

Tabu search guides the search process away from local optima by placing candidate solutions which are repeatedly visited in a list. The list is updated at each iteration of the search [66], [67]. These candidate solutions are tabu (forbidden) for a set number of iterations. This means that improvement is less likely to stagnate as these moves will be avoided. Psuedocode for the tabu search is shown in Algorithm 3.2. An initial candidate solution **S** is created using a LCH or random initialization (**1**). A tabu list is usually of length 100 and is a parameter that is problem dependent such as the population size of a genetic algorithm (**2**). The algorithm repeats until an iteration limit is reached (**kMax**) (**3**). The algorithm first applies a neighbourhood operator resulting in a solution **S'** (**3a**). If the new solution **S'** is an improvement to **S** and **S'** is not in the tabu list then set **S = S'**. If the new solution **S'** is not an improvement **S' is inserted** into the tabu list (**3b**). This way all non-improving immediate changes are avoided. The algorithm then returns a solution (**4**). Tabu search can use an aspiration criterion which allows a candidate solution in the tabu list to be selected.

1. Start with an initial candidate solution ***S***
2. Initiliase an empty tabu list
3. Repeat until **k** > **kMax**:
    a. Apply neighbourhood operator resulting in neighbouring solution **S'**
    b. **If S'** is an improvement on **S** and **S'** is not in the tabu list
        S = **S'**
        Else
            Insert **S'** into tabu list
    c. k++
4. Return best found solution

Algorithm 3.2 Tabu search [66]

3.2.3 Simulated annealing

Simulated annealing is based on the annealing process of metallurgy [68], [69]. The process of annealing removes the defects from metal through heating and cooling. The algorithm accepts all improving or equal changes in the solution space and will decrease the probability of requiring an improving or equal solution for each iteration, that if no improving or equal

solution is found, another area of the search space will be considered. This process imitates the cooling of metals. The pseudocode for the simulated annealing algorithm is presented in Algorithm 3.3. Initially simulated annealing starts with a candidate solution (**S)** (**1**). The algorithm runs until an iteration limit (**kMax**) is reached (**3**). A temperature value (**T**) is set (**3a**). The temperature is calculated based on the number of iterations (**k**) over the maximum number of iterations (**kMax**). The next step is to apply a neighbourhood operator to **S** resulting in a solution **S'**. **S'** is accepted if it is less than or equal to **S**. It is also accepted if a randomly generated probability **P** is above a set threshold. The probability for accepting a solution is based on the current solution (**S)**, the neighbourhood change to the current solution (**S'**) and the temperature (**T**). The equation for this probabilistic acceptance can be seen calculated in the Else If statement at (**3c**). It is common to include a restart mechanism to try and avoid local optima. An example restart mechanism is given in (**3e**) where a threshold will be reached if the best solution is not changed for a number of iterations. In this case if the threshold is reached the current solution **S** is replaced by the previous best solution **SBest**.

1. Start with an initial candidate solution **S**
2. **k** = 0
3. Repeat while **k** < **kMax**
    a. T = k / kMax
    b. Apply neighbourhood operator resulting in neighbouring solution **S'**
    c. **If S'** <= **S**
        S = S'
       Else If exp(-(**S'** - **S**)/**T** > **P**
        S = **S'**
    d. If S' < S
        SBest = S
       Else
        n++
    e. **If** n > nThreshold
        S = SBest
    f. **k**++
4. Return best found solution

Algorithm 3.3 Simulated annealing [68]

3.2.4 Great deluge

Great deluge is an algorithm based on the idea of floods where a water level (**B**) is updated as a threshold for accepting new solutions [70]. The algorithm is similar to the simulated annealing algorithm but it instead accepts solutions based on fixed decremented steps. The pseudocode for the great deluge is presented in Algorithm 3.4. The algorithm begins with an initial candidate solution **S** (**1**). The only parameter that needs setting is the deluge value **D,** a fraction of the Boundary level **B**, used to gradually lower the boundary level **B**. **B** the boundary level is set to the objective value of the solution **S** but is only changed by the step (**3d**). The algorithm repeats until the maximum number of iterations is reached (**kMax**) (**3**). Firstly a neighbourhood operator is applied to **S** resulting in a new solution **S'**(**3a**). **S'** is accepted if it is less than or equal to **S,** or if **S'** is less than or equal to **B**. If the solution **S'** is not accepted, a counter **n** is incremented (**3b**). If the counter **n** exceeds a threshold (**nThreshold**) then the algorithm exits (**3c**). Then the boundary level **B** is updated (**3d**) and the iteration level **k** is incremented (**3e**). When the algorithm exits a solution is returned.

1. Start with an initial candidate solution ***S***
2. Set deluge value **D**
3. Repeat while **k** < **kMax**
    a. Apply neighbourhood operator resulting in neighbouring solution **S'**
    b. **If S'** <= **S** or **S'** <= **B**
        S = **S'**
       Else
        n++
    c. **If** n > nThreshold
        exit
    d. B = B - D
    e. **k**++
4. Return best found solution

Algorithm 3.4 Great deluge [70]

3.2.5 Variable neighbourhood search

Variable neighbourhood search (VNS) is made up of the processes of shaking, local search and neighbourhood change [71]. A VNS algorithm can be seen in Algorithm 3.5. Firstly an initial solution is created either by using a LCH or randomly (1). The algorithm uses a set of neighbourhood operators (2). The algorithm repeats until the neighbourhoods have all been explored. Shaking is a change to the current solution **S**. This change can be random or as shown in the algorithm as a candidate solution generated using the current neighbourhood; this candidate solution does not have to be accepted but is an attempt to escape local optima (3a). Next the current neighbourhood operator is applied iteratively to the candidate solution (3c). If the solution is an improvement the neighbourhood explored resets to the first neighbourhood otherwise the neighbourhood is changed (3d). For example for solving the travelling salesman problem a VNS may switch from using 2-opt to 3-opt as the neighbourhood operator. Finally the algorithm returns the best solution it has found (4).

1. Start with an initial candidate solution ***S***
2. Select a set of neighbourhood operators (1...**kMax**)
3. Repeat while **k** <= **kMax**
    a. **S'** = Shaking, to a randomly produced solution from Neighbourhood $\mathbf{N_k}$
    b. **If** S' < S
        S = S'
    c. Perform local search applying neighbour $\mathbf{N_k}$ to solution **S** resulting in neighbouring solution **S'**
    d. **If** S' < S
        S = S'
        k = 1
       Else
        k = k +1
4. Return best solution found

Algorithm 3.5 Variable neighbourhood search [71]

3.2.6 Harmony search

Harmony search was inspired by the improvisation process of musicians. In the harmony search algorithm variables are treated as musicians playing notes. The notes represent different values a variable may be assigned and with each play the value is changed [72]. Harmony search is an attempt to heuristically and stochastically find the global optimum of a number of decision variables for a given problem domain. Harmony search is similar to other population based methods such as genetic algorithms [73]. The basic harmony search algorithm can be seen in Algorithm 3.6. Each candidate solution in harmony search is a set of decision variables, these are initialised randomly and stored in a harmony memory which has a size of HMS (harmony memory size) (**1**). A new solution is created by looping through each possible decision variable (**2**). Each decision variable is chosen either by selecting a random value from an existing solution in harmony memory selected by a probability called, Harmony Memory Considering Rate (HMCR). The selected value is then changed by a probability called Pitch Adjusting Rate (PAR) as a slight modification (**2a**), otherwise a random value is chosen using probability (1-HMCR) (**2b**), this is called pitch adjustment. The search attempts to find combinations of values based on experience using HMCR and PAR, or randomness that are optimal. If the new solution **S'** is better than the worst solution in the harmony memory, it replaces the worst solution (**3**). Repeat steps **2** and **3** until a criteria to stop generating new solutions is reached (**4**). Finally the best solution found is returned by the algorithm (**5**). Harmony search generates a new solution at each iteration of the algorithm stochastically using information stored in the harmony memory where as a genetic algorithm shares information given by two parents resulting in new offspring being added to the population [74]. Harmony search is similar to evolutionary algorithms as it maintains a number of solutions which are used in part to create new solutions and replaces the weakest in the harmony memory similar to replacing the weakest in a population of solutions similar to a steady-state control model [75].

1. Create a harmony memory of randomly generated solutions $\mathbf{S_{0\text{-}HMS}}$.
2. Generate a new solution **S'**, for each decision variable:
    a. **If** probability HMCR is met

Use a value of a solution in the harmony memory with. Change the value of the selected value with probability PAR.

    b. **Else** use a random value (with a probability of 1-HMCR) for this decision variable.
3. **If S'** is better than the worst solution in the harmony memory $\mathbf{S_w}$.

$$S_w = S'$$

4. Repeat 2 to 4 until a termination criteria is met.
5. Return best solution found

Algorithm 3.6 Harmony Search [72]

3.3 Summary

This chapter presents currently used methods for solving combinatorial optimization problems. The mathematical methods including, integer linear programming, constraint programming and branch and bound are presented and discussed. Secondly meta-heuristic approaches are presented these include: hill climbing, tabu search, simulated annealing, great deluge, variable neighbourhood search and harmony search. This chapter provides necessary details to understanding subsequent chapters.

Chapter 4 Nurse Rostering

This chapter introduces the nurse rostering problem. A description of the problem and brief background is given. The modeling of the problem is covered and available benchmark sets are described. A state of the art review is presented and a critical analysis of the nurse rostering problem is given.

4.1 The nurse rostering problem

The nurse rostering problem (NRP) is a scheduling problem concerned with optimizing the working schedules of nurses. The terms nurse scheduling, hospital personnel scheduling or personnel scheduling are often used as synonyms for nurse rostering. In this study nurse rostering will be defined as: the allocation of shifts to nurses for the duration of a scheduling period with the goal being to minimize a set of given constraints [76]–[78]. A shift type is a time period of a day determined by the hospital. A nurse not working is scheduled as having a free shift type for that day. A nurse working a specific shift type is considered working. A shift assignment refers to the allocation of a shift type and a day to a nurse. A nurse is an employee who works for the hospital on a contract basis e.g. a part time contract is when a nurse works 50% of the time. A contract is an agreement between the hospital and the nurse, a contract is a collection of guidelines as to how much the nurse should be working. A nurse may have a skill type that qualifies them to work a specific shift type. A scheduling period is the number of days that the roster covers. The term roster shall refer to the complete list of nurses and their shift assignments. The term schedule will be used to refer to an individual nurse's list of shift assignments. The NRP is always subject to a set of constraints to be optimized [77]–[79].

A nurse roster is a two-dimensional matrix, where the rows represent the nurses and the columns the days of the week. Each cell contains the shift that the nurse is allocated on the particular day. An example nurse roster is given in Figure 4.1. In the example 5 nurses have been assigned shifts over 4 days. For example nurse 1 has a shift E, which represents a time period in the morning, on Thursday, Friday and Sunday. Nurse 5 has a shift N, which represents a shift during the night, on Thursday and Friday. Nurse 2 has a shift L which in this example will represent the period in between morning and night shifts. The slots with no assigned shift e.g. Nurse 3 on Friday, is a free shift. Most nurse rostering problems deal with scheduling periods that are four weeks in length. The number of shift types e.g. E, L, N, is generally three to five. The number of nurses is usually in the range of 10 to 50 nurses. It should be noted that real world nurse rostering problems differ from one hospital to the next and from country to country.

Nurse rostering is a proven NP-hard problem [81] and is considered more complex than the travelling salesman problem[82]. The difficulty of the problem makes it an interesting area of research for optimization techniques. For example an NRP with 10 nurses, 4 shift types over a 4 week scheduling period has a potential search space of 10^{277}. Developments in nurse rostering may impact other personnel scheduling and timetabling problems [83].

Early studies dealt with developing mathematical models that would be used to assist in the scheduling of nursing staff. Miller et al. [84] used a formulation based on the number of required personnel per day to come up with a mathematical model of the nurse rostering problem. Miller et al. proposed a cyclic coordinate descent algorithm to solve the nurse rostering problem. This study compared the rosters produced to those created manually by the hospital finding that this approach was superior. Miller et al. [84] report the algorithm was

implemented in hospitals in the United States and Canada. Nurse rostering in the real world is mostly done by a head nurse who either manually creates the forecasted schedule or by self rostering practices where nurses choose their own working periods. Advancements in computing power allowed for more research in automated algorithms to be developed [77], [85].

Day → / Nurses ↓	Thursday	Friday	Saturday	Sunday
1	E	E		E
2	L	E	L	E
3	E		E	E
4	E	L	N	E
5	N	N		
n				E

Figure 4.1 Example nurse roster

Automated nurse rostering is generally concerned with generating an entire roster. Automated rostering solutions have mostly been successfully adopted in European hospitals [86]. Self rostering provides nurses with autonomy and satisfaction, as their personal preferences have been accounted for to a greater degree. Automated rostering may not adequately meet the needs of the nursing staff if the constraint weighting is not appropriate [77], [86]–[88]. It can be argued that self rostering is not always applicable. Self rostering introduces the possibility of creating unfair rosters [77], [89].

Kellogg et al.[86] surveyed fifteen automated systems found in literature that were implemented in hospitals. Of the fifteen systems it was not known whether five of the systems are still used, while one is partially in use. Six of the surveyed automated systems were found to still be in use with only two not in use; one was found to be partially in use. Of those still in use, over one-hundred-and-ninety hospitals use automated systems for scheduling, with only two hospitals known to have abandoned automated systems. Ten of the systems used meta-heuristics, four used mathematical methods and one was a decision support system.

The nurse rostering problem is generally solved with the aim of minimizing an objective function based on constrained optimization, or in other words the nurse rostering problem is modeled as a constraint optimization problem (COP), where the optimization of constraints is the goal. The objective function is a sum of the hard and soft constraints violations. Hard constraints must not be violated in order for a roster to be feasible. Soft constraints do not contribute to feasibility but quality and the desired goal is to minimize the number of soft constraints violated [90]. Once hard constraint violations have been eliminated, the problem is then a matter of minimizing soft constraint violations. Constraints which are not met result in a violation being counted. Soft constraints between problem instances will generally have different value weightings for incurring penalties. Each hospital and or ward will have different priorities for which constraints are more important to satisfy and minimize. For example a hospital with a low number of nurses will value minimizing staff having days off. Additionally soft constraints for some problem instances may not be necessary and thus can be weighted as 0. The soft constraints can be divided into work contract and nurse preference constraints. Work contract constraints are decided by the contract a nurse is working and nurse preference due to a nurse's decisions e.g. To not work on Thursdays.

Nurse rostering problems are generally considered highly constrained as feasible solutions are restricted by hard constraints [91]. It is also considered difficult and often impossible to satisfy all the constraints of a nurse rostering problem [78]. Further detail on constraints for the nurse rostering problem is given in section 4.2.

4.2 Benchmark sets of the nurse rostering problem domain

A common problem in combinatorial optimization is the availability of problem instances to allow for the comparison of the effectiveness of different methods. For example if a method is used to solve an instance of the nurse rostering problem that is not publicly available, it will not be possible to compare the effectiveness of this approach to other methods developed by different researchers. To overcome this researchers have created benchmark sets to compare and evaluate new and existing methods in solving the nurse rostering problem. Benchmark sets can consist of problems that are either real world or artificial. A real world data set is based upon data from an actual hospital. An artificial data set is generated by an algorithm designed to automate the creation of problems similar to real world data but based on user input. For example, the Nottingham benchmark a set [92] is based on real world data. In contrast NSPLib [93] (Nurse Scheduling Library) is a benchmark set that was generated and as such the instances are considered artificial [94]. NSPLib is not commonly used for research on the nurse rostering problem currently.

Nurse rostering involves minimizing constraint violations. The term coverage constraint is used to refer to the number of required shift assignments that should be worked each day of the scheduling period. The coverage constraint is to make sure that each shift type on a given day is allocated a sufficient number of nurses for the given shift type. In some nurse rostering problems this is not considered a hard constraint. The next constraints that should be met are those of the contracts that the nurses have with the hospital. Satisfying individual nurse preferences is also important [95] and these are referred to as personal preferences. A nurse requesting a day off or a specific shift assignment is an example of personal preferences. In this study the benchmark dataset that is used for experimentation is that used for the first international nurse rostering competition 2010 (INRC2010) [79], [96]. This was a competition created to promote research in the domain of nurse rostering given the success of the

international timetabling competitions (ITC2002 and ITC2007) to further advance educational timetabling [97]. The following subsections describe the INRC2010 benchmark data set and the Nottingham benchmark data set.

4.3 The first international nurse rostering competition 2010 (INRC2010)

This data set is a real-world data set that was developed with the intent to further reduce the gap between research and practice. Nurse rostering is a complicated problem domain and as such not all aspects of the real world problem are included. Constraints from modern hospitals were incorporated in the creation of this benchmark set.

The competition itself had 3 difficulty tracks. These were sprint, medium and long. Sprint instances were meant to be solved in 8 seconds, Medium in 8 minutes and Long were allowed up to 10 hours. A benchmarking tool was provided to determine the time limitation for specific computer configurations, the times above are given using a 3.2 GHz processor.

The hard constraints for the INRC2010 problem instances for all three tracks are:

- A nurse may only work one shift per day.
- Each shift must be assigned the required number of nurses.

Work contract constraints:

- Unwanted shift patterns: A sequence of shift types over consecutive days which are undesirable. For example a nurse working the shift type 'L' followed by 'E' would incur a soft constraint violation but a nurse working the shift type 'L' followed by 'L' would not. Unwanted shift patterns can be specified for specific consecutive days e.g. Friday, Saturday and Sunday.
- Alternative skill: A nurse is penalized for not working a shift where they have the skill required to work that shift. A free shift does not incur a soft constraint violation. Every nurse incurs a soft constraint violation for every shift that nurse works for which they do not have the required skill.
- Maximum assignment: A maximum number of shifts should be assigned to each nurse for the scheduling period. Nurses can be assigned a number of shifts during the scheduling period, up to the length of the scheduling period. Exceeding the maximum set by the contract incurs a soft constraint violation.
- Minimum assignment: The minimum number of shifts should be assigned that should be worked for the scheduling period. Nurses can be assigned a number of shifts during the scheduling period, up to the length of the scheduling period. A nurse assigned a number of shifts less than the minimum set by the contract incurs a soft constraint violation.
- Maximum consecutive working days: The maximum number of consecutive days that should be worked by a nurse. A nurses' schedule can be made with any combination of shifts worked for a nurse. The contract will specify a preferred number of maximum consecutive working days. Exceeding the maximum consecutive working days will incur a soft constraint violation.
- Minimum consecutive working days: The minimum number of consecutive days that should be worked by a nurse. A nurses' schedule can be made with any combination of shifts worked for a nurse. The contract will specify a preferred number of minimum consecutive working days. Too few consecutive working days will incur a soft constraint violation.

- Maximum consecutive free days: The maximum number of consecutive free days that can be allocated to a nurse. The contract specifies a maximum number of days each nurse is allowed to take off consecutively. A nurse will incur a soft constraint violation if too many days in a row are not worked.
- Minimum consecutive free days: The minimum number of consecutive free days that can be allocated to a nurse. The contract specifies a minimum number of days each nurse is allowed to take off consecutively. A nurse will incur a soft constraint violation if too few days in a row are not worked.
- Two free days after a night shift: It is preferable for nurses to have two free days after working a night shift. If a nurse has a night shift followed by a working day, a soft constraint violation is incurred.
- Maximum consecutive working weekends: The maximum number of consecutive weekends is measured only on weekends where the nurse is working a shift. The contract specifies a maximum number of consecutive working weekends. A nurse exceeding the maximum number of consecutive working weekends incurs a soft constraint violation.
- Minimum consecutive working weekends: The minimum number of consecutive weekends is measured only on weekends where the nurse is working a shift. The contract specifies a minimum number of consecutive working weekends. A nurse exceeding the minimum number of consecutive working weekends incurs a soft constraint violation.
- Maximum number of working weekends: The maximum number of weekends that can be worked by a nurse, where the nurse is working a shift. The contract specifies the maximum number of weekends that can be worked by a nurse. Exceeding the maximum number of working weekends incurs a soft constraint violation.
- Minimum number of working weekends: The minimum number of weekends that can be worked by a nurse, where the nurse is working a shift. The contract specifies the minimum number of weekends that can be worked by a nurse. Working less than the minimum number of working weekends incurs a soft constraint violation.
- Complete weekends: A soft constraint violation is incurred for working an incomplete weekend, a weekend is incomplete if one day in the weekend is not worked. If all days in the weekend are not worked no soft constraint violation is incurred.
- Identical shift types over weekends: A soft constraint violation is incurred if a nurse is allocated different shift types over a weekend. Days with no shifts scheduled are not penalized.

Nurse preference constraints:

- Requested day on: A specific nurse may request working on a particular day in the scheduling period. If the nurses' schedule includes not working on the requested ‘day on’, a soft constraint violation is incurred.
- Requested day off: A specific nurse may request not to work on a particular day in the scheduling period. If the nurses' schedule includes working the requested ‘day off’, a soft constraint violation is incurred.
- Requested shift on: A specific nurse may request working a specific shift type on a specific day in the scheduling period. If the nurses' schedule includes not working the requested shift type on a specific day, a soft constraint violation is incurred.
- Requested shift off: A specific nurse may request not working a specific shift type on a day in the scheduling period. If the nurses' schedule includes working the requested shift type on a specific day, a soft constraint violation is incurred.

Each problem instance in Table 4.1 uses a subset of the above soft constraints. Furthermore, the weighting of each soft constraint differs for each problem instance.

Table 4.1 shows each problem instance that was created for the competition. In this table the number of nurses and the shift types that must be worked each day are given. The scheduling period for all instances is 28 days. The constraints that feature in some instances but not all instances are: Unwanted shift patterns (**USP**), alternative skill (**AS**), Minimum consecutive working weekends (**MinCWW**), Maximum consecutive working weekends (**MaxCWW**), No night shift before a free weekend (**NNF**), day on or off requests (**D**) and shift on or off requests (**S**).

Table 4.1 INRC2010 benchmark instance data characteristics

Instance	Nurses	Shift Types	USP	AS	MinCWW	MaxCWW	NNF	D	S
sprint_early(01–10)	10	4	Yes					Yes	Yes
sprint_late01	10	4	Yes		Yes		Yes	Yes	Yes
sprint_late02	10	3	Yes		Yes		Yes	Yes	Yes
sprint_late03	10	4	Yes		Yes		Yes	Yes	Yes
sprint_late04	10	4	Yes		Yes	Yes	Yes	Yes	Yes
sprint_late05	10	4	Yes		Yes		Yes	Yes	Yes
sprint_late06	10	4			Yes		Yes	Yes	Yes
sprint_late07	10	4			Yes	Yes	Yes	Yes	Yes
sprint_late08	10	4			Yes	Yes	Yes		
sprint_late09	10	4			Yes	Yes	Yes		
sprint_late10	10	4			Yes	Yes	Yes	Yes	Yes
sprint_hidden01	10	3	Yes		Yes	Yes	Yes	Yes	Yes
sprint_hidden02	10	3	Yes		Yes		Yes	Yes	Yes
sprint_hidden(03&04)	10	4	Yes		Yes		Yes	Yes	Yes
sprint_hidden05	10	4	Yes		Yes	Yes	Yes	Yes	Yes
sprint_hidden(06&07)	10	3	Yes		Yes	Yes		Yes	Yes
sprint_hidden08	10	4	Yes		Yes	Yes		Yes	Yes
sprint_hidden09	10	4	Yes		Yes			Yes	Yes
sprint_hidden10	10	4	Yes		Yes	Yes		Yes	Yes
medium_early(01–05)	31	4						Yes	Yes
medium_late01	30	4	Yes		Yes	Yes	Yes	Yes	Yes
medium_late02	30	4	Yes		Yes	Yes	Yes	Yes	Yes
medium_late03	30	4			Yes	Yes	Yes	Yes	Yes
medium_late04	30	4	Yes		Yes	Yes	Yes	Yes	Yes
medium_late05	30	5	Yes	Yes	Yes	Yes	Yes	Yes	Yes
medium_hidden01	30	5	Yes	Yes	Yes	Yes	Yes		
medium_hidden02	30	5	Yes	Yes	Yes	Yes	Yes		
medium_hidden03	30	5	Yes		Yes	Yes	Yes		
medium_hidden04	30	5	Yes	Yes	Yes	Yes	Yes		
medium_hidden05	30	5	Yes		Yes	Yes	Yes		
long_early(01–05)	49	5	Yes					Yes	Yes
long_late01	50	5	Yes	Yes	Yes	Yes	Yes		
long_late02	50	5	Yes	Yes	Yes	Yes	Yes		
long_late(03&04)	50	5	Yes	Yes	Yes	Yes	Yes		
long_late05	50	5	Yes	Yes	Yes	Yes	Yes		
long_hidden(01–04)	50	5	Yes	Yes	Yes	Yes	Yes		
long_hidden05	50	5	Yes	Yes	Yes	Yes	Yes		

4.4 Nottingham benchmarks

The University of Nottingham currently maintains a set of benchmark instances for nurse rostering [92]. These can be seen in Table 4.2. This benchmark set is based on research using real world hospital problems. The problem model used is based on the advanced nurse rostering model (ANROM) [98]. The ANROM model was adopted by 40 hospitals in Belgium replacing manual scheduling successfully [99].

Table 4.2 Nottingham benchmark instance data

Instance	Nurses	Shift types	Scheduling period
Ozkarahan14	14	2	7
Musa	11	1	14
Millar-2Shift-DATA1&1.1	8	2	14
LLR	27	3	7
Azaiez	14	2	28
GPost & GPost-B	8	2	28
QMC-1& QMC-2	19	3	28
WHPP	30	3	14
BCV-3.46.2	46	3	26
BCV-3.46.1	13	4	29
SINTEF	24	5	21
ORTEC01 & ORTEC02	16	4	31
ERMGH	41	4	48
CHILD	41	5	42
ERRVH	51	8	48
HED01	20	5	31
Valouxis-1	16	3	28
Ikegami-2Shift-DATA1	28	2	30
Ikegami-3Shift-DATA1, Ikegami-3Shift-DATA 1.1 & Ikegami-3Shift-DATA 1.2	25	3	30
BCDT-Sep	20	4	30
MER	54	12	48

The hard constraints for this benchmark set are:

- A nurse may only work one shift per day.
- Each shift must be assigned the required number of nurses.
- Required skill: The nurse assigned to a shift must have the skill required to work that shift.

The problem is then a matter of minimizing weighted soft constraints. As in the case of the INRC2010 benchmark set, these can be divided into the work contract and nurse preferences:

Work contract:

- Minimum time between two assignments: A nurse is meant to work a single shift per day. However, two shifts may be on consecutive days but only a few hours apart. Therefore a minimum time must elapse before a new shift can be worked or a soft constraint violation is incurred.
- Alternative skill: A nurse is penalized for not working a shift where they have the skill required to work that shift. A free shift does not incur a soft constraint violation. Every nurse incurs a soft constraint violation for every shift that nurse works which they do not have the required skill.
- Maximum assignment: A maximum number of shifts assigned to each nurse for the scheduling period. Nurses can be assigned a number of shifts during the scheduling period, up to the length of the scheduling period. Exceeding the maximum set by the contract incurs a soft constraint violation.
- Minimum assignment: A minimum number of shifts assigned that should be worked for the scheduling period. Nurses can be assigned a number of shifts during the scheduling period, up to the length of the scheduling period. A nurse assigned a number of shifts less than the minimum set by the contract incurs a soft constraint violation.
- Maximum consecutive working days: A maximum number of consecutive days that

should be worked by a nurse. A nurses' schedule can be made with any combination of shifts worked for a nurse. The contract will specify a preferred number of maximum consecutive working days. Exceeding the maximum consecutive working days will incur a soft constraint violation.

- Minimum consecutive working days: A minimum number of consecutive days that should be worked by a nurse. A nurses' schedule can be made with any combination of shifts worked for a nurse. The contract will specify a preferred number of minimum consecutive working days. Too few consecutive working days will incur a soft constraint violation.
- Maximum consecutive free days: A maximum number of consecutive free days that can be allocated to a nurse. The contract specifies a maximum number of days each nurse is allowed to take off consecutively. A nurse will incur a soft constraint violation if too many days in a row are not worked.
- Minimum consecutive free days: A minimum number of consecutive free days that can be allocated to a nurse. The contract specifies a minimum number of days each nurse is allowed to take off consecutively. A nurse will incur a soft constraint violation if too few days in a row are not worked.
- Maximum number of hours worked: The maximum number of hours is a limit on how much a nurse should work based on the specific contract. A soft constraint violation is incurred if the hours worked by a nurse are greater than the maximum number of hours assigned to the nurse's contract.
- Minimum number of hours worked: A minimum number of hours must be worked based on the specific contract. A soft constraint violation is incurred if the hours worked by a nurse are less than the minimum number of hours assigned to the nurse's contract.
- Maximum number of assignments per day of week: For each nurse, each day of the week has a limit on the number of assignments that may be scheduled. A soft constraint violation is incurred if the limit on shift assignments for a specific day of the week is exceeded. This constraint can be used to enforce that employees get at least one free week day.
- Maximum number of assignments per shift type: A limit can be imposed on the number of assignments that should be scheduled for a shift type. A soft constraint violation is incurred if the maximum number of shift assignments for a specific shift type exceeds the limits placed on the particular shift type for the scheduling period. This is usually a low limit as it is used to encourage shift diversity.
- Maximum shift types per week: Each shift type has a limit as to how many times it may be worked in a week. If this constraint is present a soft constraint violation would be incurred when a shift type is scheduled over the maximum for a specific shift type in the period of a week.
- Number of consecutive shift types: Each shift type has a limit as to how many times it should be repeated in a schedule. If a nurse were to exceed the limit on working a shift type e.g. the E shift type worked 8 days in a row and the maximum being 6, a soft constraint violation would be incurred for exceeding the limit by 2 consecutive shift types.
- Two free days after a night shift: It is preferable to have two free days after working a night shift.
- No night shift before a free weekend: A night shift should not be assigned the day before a nurse’s free weekend.
- Maximum consecutive working weekends: The maximum number of consecutive weekends where the nurse is working a shift. The contract specifies a maximum number of consecutive working weekends. A nurse exceeding the maximum number of consecutive working weekends incurs a soft constraint violation.
- Minimum consecutive working weekends: The minimum number of consecutive weekends where the nurse is working a shift. The contract specifies a minimum number

of consecutive working weekends. A nurse exceeding the minimum number of consecutive working weekends incurs a soft constraint violation.

- Maximum number of working weekends: The maximum number of weekends that can be worked by a nurse, where the nurse is working a shift. The contract specifies the maximum number of weekends that can be worked by a nurse. Exceeding the maximum number of working weekends incurs a soft constraint violation.
- Minimum number of working weekends: The minimum number of weekends that can be worked by a nurse, where the nurse is working a shift. The contract specifies the minimum number of weekends that can be worked by a nurse. Working less than the minimum number of working weekends incurs a soft constraint violation.
- Complete weekends: A soft constraint violation is incurred for an incomplete weekend, a weekend is considered incomplete when one day is not worked, if no shifts are worked no soft constraint violation is incurred.
- Identical shift type during weekends: A soft constraint violation is incurred for working different shift types over a weekend. Days with no shift scheduled are not penalized.
- Maximum number of assignments on bank holidays: This constraint takes into consideration the previous scheduling period for each nurse, such that each nurse is penalized depending on bank holidays worked in the previous scheduling period as well as the current scheduling period. The intent is that hospitals prefer limiting bank holiday assignments. A nurse with a number of assignments on bank holidays that exceeds the maximum incurs a soft constraint violation.
- Shift type succession: Penalizes specific consecutive shift types e.g. a shift type E may not be followed by a shift type D. Having an E shift type followed by a D shift type for a nurse schedule would incur a soft constraint violation.

Nurse preferences:

- Requested day on: A specific nurse may request working on a particular day in the scheduling period. If the nurses' schedule includes not working the requested day on, a soft constraint violation is incurred.
- Requested shift off: A specific nurse may request not working a specific shift type on a day in the scheduling period. If the nurses' schedule includes working the requested shift type on a specific day, a soft constraint violation is incurred.
- Requested assignments: Nurses may request a specific shift assignment. If this shift assignment is not allocated a soft constraint violation is incurred.
- Tutorship: Some nursing staff may not be allowed to work alone and therefore it is required to assign them a tutor. Nursing staff that require tutorage but are not allocated a shift assignment with another nurse will incur a soft constraint violation.
- Nurses not allowed working together: Nurses may request being scheduled apart from other nurses. Rosters where shift assignments have nurses working with nurses whom they are not supposed to work with will incur a soft constraint violation.

4.5 State of the art in nurse rostering

This section presents literature that has contributed to the field of nurse rostering and the state of the art of nurse rostering. This section has been divided into mathematical approaches and meta-heuristic approaches.

4.5.1 Mathematical based approaches

This section reviews literature which has focused on solving the nurse rostering problem using approaches that make use of mathematical techniques. This includes studies modeling the problem using integer linear programming and constraint programming and studies that

specifically use exact mathematical methods for example branch and bound. A description of these mathematical approaches can be found in Chapter 3 section 3.1.

Valouxis et al. [100] use a two-phase integer programming approach. The first phase is used to assign work requirements to each nurse. This consists of randomly assigning a schedule for each week and ignoring constraints that depend on evaluating the shift type. Three operators are used:

- Cut at one day and interchange – A day *d* is selected and two partial rosters from two random nurses are created for each nurse. The first is a roster up to day *d* and the second is a roster from day *d+1* to the end of the scheduling period. The partial rosters are combined to form a new roster for each nurse.
- Cut at two days and interchange – Two days are selected and 'Cut at one day and interchange' is applied to each day.
- 2-Opt procedure – Swaps one or more shift assignments from a nurse, with every other nurse. Overall improvements to the roster are accepted.

The second phase assigns specific shift types to nurses. The shift assignment is random and performed over a 3 day period for the working shifts assigned to each nurse in phase 1. Phase 2 is repeated until no improvement is found. These two phases are repeated until the time limit is exceeded. This approach was the overall winner of the first international nurse rostering competition (INRC2010).

Santos et al. [101] used a variety of integer programming techniques such as mixed integer programming applied to the INRC2010 benchmark set. A greedy algorithm is used to create the initial roster. The greedy algorithm attempts all possible shift assignments for each nurse, keeping only the shift assignment that gives the lowest number of soft constraint violations. A heuristic search is used to generate integer programming sub-problems to be solved by CPLEX[102]. CPLEX uses the simplex method to solve the generated sub-problems. The following neighbourhoods are used by the heuristic:

- Fix days – A number of days are fixed and cannot be changed. A range of days is selected and at each iteration changes.
- Fix shifts – Each iteration results in a selected shift type that can be changed and nurses with those shifts will be the attempted area of this search. This neighbourhood is exited once all iterations have been performed.

These neighbourhoods create sub-problems which are solved by CPLEX. This approach produced improvements to the best known results at the time of publication.

Burke and Curtois [57] used two methods for the first nurse rostering competition 2010. The first was an ejection chain variable depth search, this was used to solve the sprint instances. The second was a branch and price approach. The initial solution was produced by applying variable depth search for 5 seconds. Then the problem was solved using branch and price using the COIN-OR simplex method [103]. This approach ranked fourth in the sprint track, second in the medium track and second in the long track. Burke and Curtois [57] extended the research and applied the branch and price algorithm to the Nottingham benchmarks. The algorithm was found to find the optimal solution for the majority of the benchmark set.

4.5.2 Meta-heuristic approaches

This section presents various meta-heuristics used to solve the nurse rostering problem. These include: variable neighbourhood search (VNS), tabu search and iterated local search.

Burke et al. [104] used a variable neighbourhood search (VNS) for the nurse rostering problem. The approach was tested on the ORTEC instance in the Nottingham benchmark set. The VNS

makes use of two neighbourhood operators:

- Swap shift assignment with free shift – Swap a nurse's shift type with a nurse with a free shift on the same day.
- Swap two shifts – Swaps the shifts of two nurses with assigned shifts on the same day.

The VNS was shown to perform better than a commercial package using a genetic algorithm.

Frøyseth et al. [76], [105] present an iterated local search approach to solve the nurse rostering problem for Swedish and Norwegian hospitals. The algorithm consists of two phases, construction and improvement. The construction phase is an algorithm that attempts to assign shifts based on what is considered important, what is the hardest shift type to assign and which employee it is best for it to be assigned to. The improvement phase uses a diversification mechanism. The diversification mechanism removes a number of nurses' shifts. Then the diversification mechanism enters the construction phase and iterated local search is used to create a valid roster. The neighbourhood operators used by the iterated local search are:

- Swap two shifts – Swaps two shifts of two nurses on a specific day.
- Swap three shifts of three nurses – Swaps three shifts of three nurses on a specific day.
- Swap two shifts of two nurses on two days – Swaps two shifts of two nurses on two days.

The generated rosters were evaluated by nurses with experience in manual nurse rostering. The generated rosters were considered good by staff that analysed them.

Lü and Hao [106] used a neighbourhood search approach called adaptive neighbourhood search (ANS). The adaptive neighbourhood search was entered into the first international nurse rostering competition 2010. The authors define two neighbourhood operators:

- Swap a shift type with a free shift – Swaps a shift of one nurse to a free shift on another nurse on a chosen day.
- Swap two shifts – Swaps two shifts of two nurses on a specific day and neither shift is free or the same shift type.

ANS switches between algorithms. The first algorithm is an intensive search that uses the tabu search meta-heuristic. The second algorithm attempts all feasible moves for a specific nurse on a specific day for a given neighbourhood operator. The second algorithm is applied to half the nurses selected randomly. The third algorithm selects a random set of half the nurses and considers all moves that can decrease soft constraint violations. The three algorithms use the neighbourhood operators, one swap and two swap. A variable called diversification level is changed throughout the search to decide which of the three algorithms should be used. The approach produced 12 improvements to the best known solutions for the INRC2010 benchmark set. This work used algorithms which used neighbourhood operators and it was a very effective approach. This approach was placed third for the sprint data set and fourth for the medium data set in INRC2010.

Vu et al. [107] use iterated local search in combination with tabu search. The approach was used to solve the nurse rostering problem for a Canadian hospital. Greedy shuffling and steepest descent are used to improve the solution created by iterated local search. Greedy shuffling should not be used for large problems if computation time is limited. Iterated local search makes use of four neighbourhood operators:

- Swap shift assignment with a free shift – Swaps the shift type of one nurse for a free shift of another nurse on a chosen day.
- Swap Worst-Scheduled nurse (the nurse with the most personal and working constraint violations) – Swaps a shift assignment of the worst scheduled nurse, with another nurse on a chosen day.
- Swap Worst-Scheduled nurse two days – Swaps two consecutive shift assignments of the worst scheduled nurse with another nurse's over the same two days.

- Swap Worst-Scheduled nurse three days – Swaps three consecutive shift assignments of the worst scheduled nurse with another nurse's over the same three days.

Tassopoulos et al. [108] use a two-phase stochastic neighbourhood search. This method was tested on seven of the Nottingham benchmark instances and also applied to the INRC2010 benchmark instances. The approach mainly uses an algorithm called: selective partial swap. Selective partial swap is a greedy algorithm applied to two nurses' schedules. For each day in the scheduling period, over a subset of days (where the subset of days is the current day and the current day+n, n increments until the last day in the scheduling period), the shifts between the two nurses are swapped under a set probability. Swaps are kept if they are improving or equal. This is similar to the greedy shuffle algorithm [109] but instead is stochastic due to the introduced randomness. The approach uses three algorithms:

- Successive segment swap mutation – Performs selective partial swap on a random list of all nurses. Select consecutive nurses in the list and perform 'selective partial swap'.
- Random segment swap mutation – Performs 'selective partial swap' over two random lists of all nurses. A nurse from each list is selected and their shifts are changed using 'selective partial swap'.
- Selective day swap mutation – Attempt swaps of a randomly selected nurse with all possible nurses for each day of the scheduling period. Keeps improving or equal swaps.

The algorithm is applied to a population of solutions; the population size used was 2. The first phase executes successive segment swap mutation for a set number of cycles after randomly creating an initial roster. The second phase for each individual in the population executes 'selective day swap mutation' then 'successive segment swap mutation' and then 'Random segment swap mutation'. Until a termination criterion is met, theses were a maximum number of iterations and a maximum number of iterations were the fitness remains unchanged. This approach found an improvement for the Nottingham benchmark instance HED01 and found improvements for medium_hidden01, medium_hidden03 and medium_hidden04.

Nonobe [110] used an existing tabu search algorithm [111] to solve the nurse rostering problem. The approach formulates the nurse rostering problem as a constraint optimization problem and then applies the tabu search algorithm. The tabu search features an adaptive tenure for the tabu list and an aspiration criterion. This algorithm had three stopping criteria: maximum number of iterations, reaching a number of iterations where no improvement has occurred and a solution with no soft constraint violations. The tabu search uses two neighbourhoods; a 'shift' neighbourhood which changes the variable values in the solution and a 'large' neighbourhood that includes the 'shift' neighbourhood and a 'swap' neighbourhood. The 'swap' neighbourhood swaps variables of different values. The 'swap' neighbourhood increases for each iteration of the tabu search. This approach was ranked second in the sprint track, third in the medium track and fourth in the long track for the first international nurse rostering competition.

Hadwan et al. [112] applied the harmony search algorithm to the nurse rostering problem. A harmony memory is represented by a two dimensional matrix. Rows represent a set of rosters. Columns represent the variables that are the shift types and nurses. To satisfy hard constraints, only valid weekly successive shift types were generated. These valid weekly shift types are randomly chosen from to create the roster for each nurse. The soft constraint violations of each roster are stored in the harmony memory. Rosters are then operated upon by three operators to create new rosters. These are: memory consideration, random consideration and pitch adjustment. Memory consideration is based on a probability to select a variable from the harmony memory. Random consideration is based on the inverse probability of memory consideration and randomly changes the value of the variable. Pitch adjustment attempts to change the value of variables based on another probability and a randomly chosen value for the amount of change (bandwidth). For pitch adjustment each only improving solutions are accepted. The algorithm was tested on data from a large hospital in Malaysia and was also

tested on the Nottingham benchmarks. This harmony search algorithm was found to perform better than a VNS [104] (on a single instance only), a shift sequence approach [113] and a memetic approach [114] but was found to perform worse than the scatter search algorithm used in [57].

Awadallah et al. [115] used a harmony search algorithm applied to the nurse rostering problem. The harmony memory is randomly generated. Pitch adjustment is applied to all nurses and each shift assignment in the nurse's roster. The pitch adjustment operator is changed to use three neighbourhood operators when changing a shift type:

- Move one shift – Swap the nurse's shift type with a random nurse with a free shift type.
- Swap two shifts – Swap the nurse's shift type with a random nurse with a different shift type.
- Do nothing – Make no changes.

These operators are used based on a probability for each. Moves made by the operators to the candidate solution were discarded if they resulted in a worse candidate solution. Memory consideration assigns shift types to nurses from those in the harmony memory based on a probability. Random consideration assigns remaining nurses shift types based in the inverse probability. If an infeasible solution is generated the process begins again. Following these operators pitch adjustment is performed. This is done for each nurse. The stopping criterion for the algorithm was a maximum number of iterations. Harmony search was not able to compete with the results from the first international nurse rostering competition 2010 but was only tested on the set of sprint instances.

Awadallah et al. extend their work in [116], where the random selection of variables in the harmony search memory is exchanged for the method 'global best'[117]. This method is reportedly from particle swarm optimization. The global best method instead selects the best value in the harmony memory instead of a random value. More neighbourhood operators are explored in pitch adjustment operator compared to [115]:

- Move one shift – Swap the nurse's shift type with a random nurse with a free shift type.
- Swap two shifts – Swap the nurse's shift type with a random nurse with a different shift type.
- Token ring move – Swap the nurse's shift type with a random shift assignment of another nurse, if the complete weekend soft constraint is violated. The selected shift assignment will also be swapped with another random nurse's shift assignment.
- Cross move – Swap the nurse's shift type with a random nurse's shift type but the shift type of the assignment is the same.
- Move weekend – Swap the nurse's shift type if this shift assignment is during a weekend, swap the shift types of the entire weekend with a randomly selected nurse's shift types during the weekend.
- Swap two days – Swap the nurse's shift type and the consecutive shift type on the following day with a random nurse's shift assignments for the same days.
- Swap three days – Swap the nurse's shift type and two consecutive shift type on the following day with a random nurse's shift assignments for the same days.

This time the harmony search algorithm is applied to the entire INRC2010 benchmark set. Random selection is shown to perform better on the sprint track from the benchmark set compared to using the global best method. This algorithm outperforms the harmony search algorithm used in [115].

Awadallah et al. [118] hybridized the harmony search algorithm with greedy shuffle as it was previously found to be effective in [119], [104], [120]. The greedy shuffle algorithm is a local search method which swaps each shift of each nurse with every shift in the scheduling period but only accepts swaps which improve the candidate solution. Greedy shuffle was applied after creating a new harmony under a certain probability. This was tested on the INRC2010

benchmark set. Harmony search using the global best method to select from the harmony memory, performed slightly better on medium instances but worse on long track instances for the INRC2010 benchmark set [118].

Awadallah et al. [121] extend the harmony search algorithm to include a hill climbing operator during the generate a new solution phase (see Algorithm 3.6). Hill climbing uses four neighbourhood operators:

- Move weekend – Swap the nurse's shift type if this shift assignment is during a weekend, swap the shift types of the entire weekend with a randomly selected nurse's shift types during the weekend.
- Move one shift – Swap the nurse's shift type with a random nurse with a free shift type.
- Swap two shifts – Swap the nurse's shift type with a random nurse with a different shift type.
- Shuffle moves – Selects a nurse with the worst roster (the highest soft constraint violations) and attempts to incrementally swap subsets of shift assignments starting from a randomly selected day until the end of the scheduling period, with a randomly selected nurse.

This hybridization saw improved performance compared to global best harmony search [116] for the entire INRC2010 benchmark instance set. The algorithm performed well on the sprint instances but struggled to achieve good results on the medium and long instances. The authors suggested that they would have improved results by implementing new neighbourhoods.

4.6 Critical analysis

When evaluating a method it is often worthwhile to have a means to compare it with others. This is difficult without a reliable benchmark set. In literature there are very little reported results using the NSPLib benchmark data set. NSPLib has a large number of problem instances; while this is great for accurate statistical tests, it is not practical to the real world researcher. This is because it is difficult to find comparisons for results of problem instances and additional time must be spent doing experiments for problem instances where comparisons could be made. The Nottingham benchmark instances are a good resource which provides examples of hospital models from various countries. The problem is only a subset of this benchmark set is used by most researchers making it difficult to compare different approaches across the entire benchmark set. The nurse rostering competition benchmark instances are a real world data set but have only two strict hard constraints compared to three in the Nottingham benchmark instances. The main advantage is that new methods to solve the nurse rostering problem are being reported using these benchmark instances. There are also a number of reported results from the first international nurse rostering competition INRC2010. While there is some inconsistency in what instances have been reported, it is still possible to compare. It is also useful that one of the reported results is a selection perturbative hyper-heuristic. For these reasons the INRC2010 benchmark set will be used in this study.

Integer programming and exact methods such as branch and price perform very well when applied to the nurse rostering problem. These have the advantage of being very reliable but have the disadvantage of being potentially computationally expensive, especially branch and price. Flexibility of the model used for integer programming approaches is also not guaranteed. The integer programing approaches presented have used local search algorithms in addition to mathematical methods. Meta-heuristics with a focus on neighbourhood search, such as the adaptive neighbourhood search (ANS) [106] and the two-phase stochastic neighbourhood search [108] have performed competitively on the INRC2010 benchmark set in terms of finding minimum values. The two-phase stochastic neighbourhood search has shown very good performance for the nurse rostering problem. The performance of the two-phase stochastic

neighbourhood search has not been reported outside of the INRC2010 competition website [96] but it has been capable of finding and improving on the best known results for the INRC2010 benchmark instances. A main feature of this neighbourhood search is its probabilistic greedy swap algorithm, selective partial swap.

The field of nurse rostering is a domain where there are not that many new contributions, however new approaches have had success in improving benchmark results. The two-phase stochastic neighbourhood search [108] improved on and matched results obtained by state of the art integer programming techniques [101]. Most of the recent and previous research is focused on solving specific hospital problems. This makes it difficult to compare approaches. It also makes it difficult to determine if what worked for a study is effective for the nurse rostering problem overall, as often the problem instances are private.

4.7 Summary

This chapter presents the nurse rostering problem, the benchmark sets commonly used, the state of the art methods applied to solve the nurse rostering problem and finally a critical analysis of the nurse rostering problem is presented.

Chapter 5 Hyper-Heuristics

This chapter introduces the field of hyper-heuristics. Firstly an introduction to the field of hyper-heuristics is given. Then each of the four categories of hyper-heuristics, identified in literature are reviewed.

5.1 Introduction to hyper-heuristics

The field of hyper-heuristics aims to provide a problem solving methodology that has a higher level of generality than methods such as branch and bound, integer programming and meta-heuristics e.g. genetic algorithms and simulated annealing or problem specific low-level heuristics. Hyper-heuristics operate indirectly, mapping the low-level heuristics onto a candidate solution. The intent is that a hyper-heuristic can be reused in multiple problem domains. The reusability saves time compared to creating problem specific solutions. Hyper-heuristics use information gained mostly from an objective function, a measure of the candidate solution's quality.

The first reported use of the term hyper-heuristic was by Cowling et al. [122]. Hyper-heuristics were first defined as a problem independent method to choose low-level heuristics. An objective function is used to inform the hyper-heuristic to make the best choice of which low-level heuristic to choose and apply to a candidate solution. Burke et al. [123] identified the probabilistic learning technique used to combine low-level heuristics to solve the job shop scheduling problem by Fisher and Thompson [124] as an early hyper-heuristic. Other approaches that have been categorized as hyper-heuristics were also found in work involving genetic algorithms in [125]. In [125] the job shop scheduling problem was investigated. A genetic algorithm was tested to evolve low-level heuristic combinations. An individual represents a combination of low-level construction heuristics. For example the individual "abde..." would use the low-level construction heuristic represented by 'a' to schedule the job represented by 'b'. In [126] a genetic algorithm was used to solve the transportation of chickens for a company in Scotland; as in the previous study the genetic algorithm evolves a string where each character represents a low-level construction heuristic. Each low-level construction heuristic is applied sequentially to build a schedule. There are two main types of hyper-heuristics, selection and generative.

The term heuristic is a heavily overloaded term in literature. Perturbation heuristics are moves which change (perturb) a candidate solution. For example, swapping two nodes in a directed graph would be a perturbation heuristic. Construction heuristics are used to build candidate solutions – for example, adding a node to a directed graph that is the closest in distance to the previous node (nearest neighbour) would be a construction heuristic. In hyper-heuristics these two types are referred to as low-level perturbation heuristics (LPHs) and low-level construction heuristics (LCHs).

Selection hyper-heuristics are a category of hyper-heuristics where low-level heuristics are selected and applied to a candidate solution. The selection hyper-heuristic selects from a provided set of domain specific low-level heuristics. Selection hyper-heuristics can be categorized into selection perturbative and selection constructive hyper-heuristics.

Construction heuristics are used to build a solution based on simple rules. Construction heuristics are usually specific to the problem domain. An example of a construction heuristic is the saturation degree heuristic. Saturation degree is a heuristic based on graph colouring and

presented in [127] for examination timetabling. The saturation degree heuristic first calculates the number of feasible periods for each event. A feasible period is a time slot that will not incur a hard constraint violation. The event with the least feasible periods available is given priority. An example of a selection construction hyper-heuristic is that implemented by Petrovic and Qu [128]. In this work case-based reasoning was used. A database of cases of timetabling problems paired with the best two low-level heuristics to solve each problem is stored. The hyper-heuristic then uses the generated knowledge to apply the most suitable low-level heuristic to solve unseen timetabling problems.

Burke et al. [129] extended the concept of a hyper-heuristic to include the generative hyper-heuristic. This category of hyper-heuristic aims to create new low-level heuristics. As with the selection hyper-heuristics, generative hyper-heuristics can be identified in earlier studies. An example is the work by Fukunaga on the satisfiability (SAT) problem[130], [10], [131]. Generative hyper-heuristics are a type of hyper-heuristic that create new low-level heuristics for a chosen problem domain. The generative hyper-heuristic can be subdivided into construction and perturbation categories. The new low-level heuristics are created by using existing low-level heuristics and features of the problem. According to Fukunaga [130] these components used for generating new heuristics are best identified by human designers. The components used by generative hyper-heuristics are features of low-level heuristics, low-level heuristics and features of the problem used for creating new low-level heuristics. Generative hyper-heuristics most commonly use genetic programming to create construction heuristics. Most of the work on generating perturbation heuristics has been done with grammatical evolution, a variant of genetic programming [132]. The heuristics generated by these approaches can be disposable or reusable [133]. Burke et al. [134] found the size of the training set and difficulty of the training instances to be a factor affecting reusability of generated heuristics for packing problems.

The generative perturbative hyper-heuristic creates perturbation heuristics. An example of a generative perturbative hyper-heuristic is found in [134]. In [134] grammatical evolution is used to create bin-packing perturbation heuristics. These evolved heuristics are reusable.

The generative constructive hyper-heuristic creates construction heuristics, an example of a generative constructive hyper-heuristic can be found in [135]. In [135] reusable construction heuristics are generated for the knapsack problem using genetic programming. The generated heuristics outperformed human designed heuristics. The ‘heuristics’ created by generative constructive hyper-heuristics will be referred to as generated construction heuristics (GCHs) and those created by generative perturbative hyper-heuristics as generated perturbation heuristics (GPHs).

There are hyper-heuristics with no learning and hyper-heuristics with learning [123]. A hyper-heuristic that does not use the feedback provided by the objective function has no learning. An example of a hyper-heuristic without learning is a random selection perturbative hyper-heuristic that accepts all moves.

Hyper-heuristics that feature learning fall into two categories of learning: offline and online learning. The type of learning is dependent on when the learning occurs in the hyper-heuristic. Online learning refers to learning based on the current information gained during the execution of the hyper-heuristic. The selection hyper-heuristic used in [122] is an example of online learning where a choice function is used to select low-level heuristics . The choice function selects low-level heuristics is based on the feedback learnt from the effectiveness of already applied low-level heuristics. Offline learning refers to learning obtained at an earlier stage; this information is used to inform the hyper-heuristic at runtime. This is usually a set of training instances used in the learning process. The information gained is used by the hyper-heuristic during an execution with instances that have not been part of the offline learning. Chan et al.

[136] generated a decision tree to select which groups of low-level heuristics to apply based on information gained during previous executions, such as the number of suboptimal solutions found and the percentage of suboptimal moves found after using groups of low-level heuristics.

Hyper-heuristics have limited interaction with the problem domain. This is because the hyper-heuristic only receives feedback from the objective function based on the effect of the low-level heuristic on the candidate solution. When the hyper-heuristic cannot access domain specific information, this is called the domain barrier [31], [122], [137]–[140]. The domain barrier allows the hyper-heuristic to indirectly interact with the candidate solution. Hyper-heuristics use only the provided low-level heuristics and the objective function for feedback. Maintaining the domain barrier increases the generality of the hyper-heuristic. This is different from meta-heuristics that operate by searching the solution space. This generality is achieved through a mapping from the heuristic space to the solution space [123], [137].

Recently attempts to provide hyper-heuristic frameworks that generalize across a number of domains have been explored. The Hyflex framework allows for implementation of a selection perturbative hyper-heuristic that can be benchmarked across multiple domains [141]. Hyflex includes benchmark datasets and low-level heuristics for the problem domains of personnel scheduling, traveling salesman, bin packing, SAT and permutation flow shop scheduling [141]–[145]. In the Hyflex framework, the problem domains and LPHs are made available through an interface to the user. An alternate concept exists in the Hyperion framework which allows meta-heuristics and hyper-heuristics to be implemented and studied. This framework is not limited to specific problem domains. Hyperion provides existing move acceptance methods for use such as accept all moves, accept only improving, exponential Monte Carlo, simulated annealing and great deluge [146], [147]. These are provided for the purpose of exploring and analyzing the effectiveness of these move acceptance criteria with meta-heuristics and or selection perturbative hyper-heuristics. This progress in research provides proof of the main goal of hyper-heuristics, the goal of a more generalized approach for problem solving. Hyper-heuristics are searching for reliable problem independent methods that are good at generalizing across problem domains [148], [149].

In the following sections relevant literature across the four categories of hyper-heuristics will be presented. The sections will cover the methods employed, the domain the hyper-heuristic was applied to and any relevant outcomes: Firstly the selection perturbative hyper-heuristic, secondly the selection construction hyper-heuristic, thirdly the generative construction hyper-heuristic and finally the generative perturbative hyper-heuristic is presented. This order was chosen as selection perturbative hyper-heuristics are arguably the most common hyper-heuristic and it is necessary to see how selection differs between the selection constructive and selection perturbative hyper-heuristics. The generative construction hyper-heuristic is the most researched type of generative hyper-heuristic and thus should be used to influence the research that will lead to the more recent field of generative perturbative hyper-heuristics.

5.2 Selection perturbative hyper-heuristics

This section will introduce and survey selection perturbative hyper-heuristics. This section will look at relevant literature on selection perturbative hyper-heuristics.

Selection perturbative hyper-heuristics are generally comprised of a selection method and a move acceptance method [150], [151]. The selection method decides which heuristic in the provided set of LPHs to select and apply to the candidate solution. If only a single solution is improved by exploring a set of neighbourhood moves, it is called a single-point search [152]. The move acceptance method decides if the move made by a selected LPH is accepted. It is

possible to use accept all moves as a move acceptance method. Multi-point search methods when used as selection perturbative hyper-heuristics, e.g. genetic algorithms, are both heuristic selection and move acceptance. In the case of a genetic algorithm selection perturbative hyper-heuristic the process of evolution and the fitness function take care of selecting heuristics and whether or not to accept a move. The LPHs are problem domain dependent. This is because the LPHs must map to the solution space of the problem domain. The selection perturbative hyper-heuristic aims to outperform the individual LPHs [153].

Ayob and Kendall [154] investigated a hyper-heuristic that uses random heuristic selection. The hyper-heuristic was used to solve the component placement problem of an electrical circuit board. It was found to be best solved using EMCQ (Exponential Monte Carlo with counter) as the move acceptance method. The authors found that a hyper-heuristic using choice function selection and simulated annealing move acceptance performed poorly in comparison.

Bai and Kendall [155] compared two selection perturbative hyper-heuristics to solve the problem of shelf organization. One of the hyper-heuristics used random heuristic selection and the other choice function heuristic selection. Three move acceptance methods were tested for the random heuristic selection hyper-heuristic. These move acceptance methods were accept improving moves, accept all moves and simulated annealing. The move acceptance for the choice function heuristic selection hyper-heuristic was not reported. The choice function heuristic selection hyper-heuristic was based on [138]. The hyper-heuristic with random heuristic selection and simulated annealing move acceptance was found to be the best performing hyper-heuristic.

Kendall and Soubeiga [156] investigate two hyper-heuristics. One with random heuristic selection and one with choice function heuristic selection. These are tested on a project presentation scheduling problem. The move acceptance methods investigated were accept all moves and accept only improving moves. The hyper-heuristic with choice function heuristic selection outperformed the hyper-heuristic with random heuristic selection on tests which were performed on initial solutions of a poor quality. It was found that a hyper-heuristic using choice function heuristic selection performed better than the hyper-heuristic using random heuristic selection. The move acceptance methods tested were ‘accept all moves’ and ‘accept only improving moves’. Choice function heuristic selection was able to find solutions of better quality in a shorter time period.

Burke et al. [157] investigate the heuristic selection methods of choice function and random selection. A comparison is given of the move acceptance methods of simulated annealing and EMCQ (Exponential Monte Carlo with counter). EMCQ decreases the chance of accepting a solution exponentially with time but if there is no improvement the chance of accepting solutions is increased. The hyper-heuristics tested are similar to those in [154]. The hyper-heuristics were tested on the examination timetabling problem using the Carter benchmark set [158]. The choice function selection hyper-heuristic with simulated annealing move acceptance was found to be better than a random selection hyper-heuristic with EMCQ as move acceptance.

Özcan et al. [159] compare the performance of greedy random heuristic selection, reinforcement learning heuristic selection and simple random heuristic selection in a hyper-heuristic for solving the examination timetabling problem. Late acceptance was used for move acceptance. Late acceptance is similar to hill climbing and simulated annealing. Late acceptance hill climbing differs from standard hill climbing algorithms by accepting candidates by considering previous candidate solutions. The hyper-heuristics were evaluated on the Carter benchmark set [158]. Simple random heuristic selection and late acceptance were found to perform the best. This hyper-heuristic outperformed a hyper-heuristic using choice function heuristic selection and simulated annealing move acceptance [157].

McClymont and Keedwell [160] implemented a hyper-heuristic using a Markov chain for heuristic selection and move acceptance. The hyper-heuristic was tested on multi-objective test problems. The method only uses four LPHs, namely, mutation, replication and transposition. This approach was compared to a random selection hyper-heuristic using the same LPHs however the acceptance method used was not mentioned. This hyper-heuristic outperformed the random selection hyper-heuristic.

Burke et al. [161] use an ant colony algorithm for heuristic selection. This hyper-heuristic was applied to the project presentation scheduling problem. A variety of move acceptance methods were investigated. The move acceptance, accept all moves was more successful than accepting improving moves only. This hyper-heuristic was found to perform better than the hyper-heuristic using choice function selection from [156]. It produced better results than a hyper-heuristic using simple random heuristic selection. In [162] a similar ant colony hyper-heuristic is applied to the travelling tournament problem. The move acceptance used was not reported. The hyper-heuristic produced better results than integer programming, constraint programming, tabu search and ant colony optimization approaches used to solve the same travelling tournament problem instances.

Burke et al. [163] used tabu search for heuristic selection as part of a hyper-heuristic to solve two problems namely, the university course timetabling problem and nurse rostering problem. The authors do not specify the use of a specific move acceptance method. This hyper-heuristic was not changed only the low-level heuristics used were changed. The hyper-heuristic was not as good as a tailor-made genetic algorithm for the nurse rostering problem. For the university course timetabling problem, the hyper-heuristic performed better than or equal to an ant colony algorithm and a local search algorithm. It is important to note that in this study no parameters were changed for the tabu search algorithm when it was used for a different problem domain.

Kendall and Hussin [164], [165] used a hyper-heuristic similar to that of [163] to solve examination timetabling problems. The hyper-heuristic used tabu search for heuristic selection. Three variants of move acceptance methods were compared namely tabu search, hill climbing and great deluge. It was reported that great deluge acceptance performed well by directing the search towards better quality solutions. The authors note that the hyper-heuristic took up to 10 times longer than using a great deluge meta-heuristic. This can be expected because a hyper-heuristic interacts indirectly with the solution space. Hyper-heuristics are known to require more runtime in general.

Cowling et al. [31] employed a genetic algorithm using an indirect representation as a selection perturbative hyper-heuristic to solve a trainer scheduler problem. Each chromosome represents a string of selected LPHs. The LPHs are applied in sequence to a candidate solution. Evolution is achieved by using genetic operators. The genetic operators used were crossover and mutation. This work is extended in [31] to incorporate an adaptive length representation rather than fixed length by using cut and splice crossover. The adaptive length genetic algorithm hyper-heuristic was found to perform better than the individual heuristics, a genetic algorithm and the genetic algorithm hyper-heuristic in [22]. In [40] the adaptive length genetic algorithm hyper-heuristic used guided genetic operators. These operators were specialized mutation operators which removed poorly performing heuristics from longer than average individuals. The length of the average individual is kept track of during the execution. This variation also injects LPHs that are considered good into shorter than average individuals. This version of the hyper-heuristic is applied to the student project presentation scheduling problem and the geographically distributed course scheduling problem [40]. In [40] a tabu list was used to limit the use of LPHs that did not change the objective function from being called for a number of subsequent generations. This tabu criterion is only applied to the position of the LPH in the chromosome. It was found that using this tabu list did not provide better results when compared to the previous

genetic algorithm hyper-heuristic using guided genetic operators, a genetic algorithm, memetic algorithm and choice function hyper-heuristic from [156]. The authors state that the genetic algorithm using the tabu list and the choice function hyper-heuristic require less processing time compared to the genetic algorithm hyper-heuristic using guided genetic operators.

The selection perturbative hyper-heuristic is a powerful problem solving method. Selection perturbative hyper-heuristics show good generality over different problem domains. Generally a selection perturbative hyper-heuristic is more flexible than a meta-heuristic approach. Selection perturbative hyper-heuristics can more easily be applied to different variations of a problem domain and if necessary to different problem domains. The design of a selection perturbative hyper-heuristic is dependent on choosing either a good performing heuristic selection method and or a good move acceptance method.

5.3 Selection construction hyper-heuristics

Selection constructive hyper-heuristics select LCHs. Each LCH determines an allocation to the candidate solution based on its own measure. The suggested allocation is then used to construct the candidate solution [151]. The aim is to make more intelligent use of LCHs. A single LCH can be used to create an entire candidate solution. LCHs are good at creating feasible solutions. A string produced by a genetic algorithm, where each character represents a LCH is still a selection hyper-heuristic because a hyper-heuristic (the genetic algorithm) has selected the order of application for those LCHs.

Burke et al. [123] consider the work by Thompson and Fisher, and Crowston et al. [124], [166] as some of the first hyper-heuristics. This work used probabilistic learning a classification approach based on the probability of an event happening. This was used to select known job shop scheduling construction heuristics. The probabilistic learning updated the performance of the applied LCH. The associated performance of each LCH decided which construction heuristic was best to use when creating a candidate solution.

The ideas of Thompson and Fisher, and Crowston et al. [124], [166] are explored further using genetic algorithms by Fang et al. [125], [167]. Fang et al. [125], [167] used a genetic algorithm to solve the job shop scheduling problem. In this genetic algorithm each individual represents a sequence of LCHs and jobs that require allocation. This was effectively a selection constructive hyper-heuristic and searches the heuristic space. The selection construction hyper-heuristic was found to perform better than a standard genetic algorithm applied to the solution space.

López-camacho et al. [168] implement a genetic algorithm to evolve LCH strings to solve 1D and 2D bin-packing problems. A single allocation method was used to fill bins, where other similar studies let the genetic algorithm also choose the allocation method in the encoding. The evolved string of LCHs outperforms the individual heuristics.

Ross and Marin-Blázquez [169] bused XCS, an evolutionary algorithm classifier as a hyper-heuristic. This hyper-heuristic was used to select a set of construction heuristics to solve the bin-packing problem. The evolved heuristic combination was able to outperform the largest fit decreasing heuristic and the variation of Django and Finch's heuristic which was found to be the best performing heuristic. A training set of 10 instances was used. The evolved combinations found were evaluated on 223 instances. The final evolved combination solved more instances to optimality than the best individual heuristic.

Pillay [170] used an evolutionary algorithm hyper-heuristic to solve the uncapacitated examination timetabling problem. The individuals in the population are represented by strings.

Each character in the string represents a LCH. The LCHs are the graph colouring construction heuristics defined for examination timetabling [171]. The approach was tested on instances from the Carter examination timetabling benchmark set [158]. This research tested different representations for individuals in the population. These were: variable length heuristic combination, n-times heuristic combination and fixed length heuristic combination. All representations found feasible timetables. It was found that a combination of all the representations tested generalized better than each LCH.

Els and Pillay [172] used an evolutionary algorithm hyper-heuristic to solve the curriculum based university course timetabling problem. This approach was tested on a set of 14 benchmark problems from the ITC2007 benchmarks [97]. Each individual was made up of characters that map to graph colouring construction heuristics found in [171]. The size of the individuals was set to be problem dependent. The combinations evolved were analysed to find patterns in the occurrence of construction heuristics. It was found that the order and frequency of LCHs were problem dependent. The evolutionary algorithm hyper-heuristic produced feasible timetables for all runs. This was better than individual LCH performance.

Burke et al. [127] used a tabu search hyper-heuristic to solve examination timetabling problems. Graph colouring LCHs were used [171]. It was found that the tabu search works best for heuristic selection when combined with a deepest descent local search [127]. Deepest descent is applied to the candidate solution after each LPH application. Using a larger set of LCHs provided better results. The Carter examination timetabling benchmark set [158] was used to test the hyper-heuristic. The larger set of LCHs adversely affected computational time due to exploration of a larger search space. The same hyper-heuristic was also tested using the ITC2007 university course timetabling benchmark set [97]. The hyper-heuristic produced better results for one problem instance compared to previous work employing a tabu search hyper-heuristic, local search and ant colonization [173]. This work was extended in [174] by Qu and Burke investigating using local search during timetable construction. A greedy search was performed when a completed candidate solution was produced and during candidate solution construction. This resulted in improved results over those found in [127]. In [174] it was found that iterative local search and variable neighbourhood search were more effective than tabu search and the steepest descent method for the hyper-heuristic to choose LCHs. The Carter examination timetabling benchmark set [158] and the ITC2007 university course timetabling benchmark set [97] were used for testing the selection construction hyper-heuristic. This work shows the advantages of searching the heuristic search space with the solution space.

Sabar et al. [175] used four ordered lists of four selected LCH to solve examination time tabling problems. In this approach the heuristic groups are applied hierarchically similar to the approach used by Pillay and Banzhaf [176]. Each group was a permutation of selected graph colouring heuristics. The timeslots selected are chosen using roulette wheel selection. This work differs from the work of Pillay and Banzhaf [176] by using a difficulty index given by the sum of the suggested order given by each individual heuristic for each exam. The lowest difficulty index indicates the most difficult exam to be scheduled first. This approach was found to be better than using a single LCH, which was selected and applied sequentially. This approach was tested on the Carter benchmarks [158] and the ITC2007 benchmarks [97]. It was suggested that hierarchical combinations are an effective strategy when compared to sequential combinations used by hyper-heuristics generally.

Selection constructive hyper-heuristics have been shown to be effective for creating candidate solutions. It is evident that construction heuristics are very useful for providing feasible initial solutions.

5.4 Generative constructive hyper-heuristics

Generative constructive hyper-heuristics describe a hyper-heuristic approach that generates new LCHs or GCH [151]. These GCHs are found using components. Components are human identified and include features of the problem and existing LCHs. The chosen components are recombined by the hyper-heuristic to create a GCH. Generative constructive hyper-heuristics aim to automate the process of creating equal or better LCHs. This results in a new generated construction heuristic (GCH). Automating the creation of LCHs saves time when working on new problems. The studies presented here made contributions to the field.

Oltean and Dumitrescu [177] evolved GCHs for the travelling salesman problem using multi expression programming. Multi expression programming is a variant of genetic programming that uses a linear representation [178]. The best GCH that was found through evolution was compared to the nearest neighbour construction heuristic and the minimum spanning tree construction heuristic on 17 instances from TSPLIB [179]. The GCH performed better than the minimum spanning tree construction heuristic and better than the nearest neighbor LCH on 14 of the 17 instances tested.

Poli et al. [180] use a linear genetic programming approach to evolve construction heuristics. A steady state control model was used. Two different crossover operators were selected from uniformly. The resulting offspring had point mutation applied to them. The primitives used consisted of arithmetic operators applied to an accumulator. There were two accumulators used to provide the ability to store and swap values. Registers were used to input problem specific information. The GCHs were used to solve the one dimensional bin packing problem. A training set with 48 problem instances was used. The GCHs were reported to generally outperform human designed heuristics.

Dimopoulos and Zalzala [181] used genetic programming to create GCHs for a machine scheduling problem. A set of 20 training instances was used. The function set consisted of arithmetic operators. The terminal set was derived from the parameters which comprise the LCH called, Montagne [182]. These parameters were: the processing time of a job, due date of a job and sum of processing time of all jobs in the problem instance. The earliest due date and shortest processing time LCHs were included in the components used. It was reported that the GCHs were found to be equivalent to the individual LCHs in performance on test instances.

Geiger et al. [183] developed an evolutionary learning system to create GCHs. The GCHs are used to solve a machine scheduling problem. This approach is similar to that of [181] but more problem specific components are used and no LCHs are used. Arithmetic functions and a conditional function were used to form a GCH. The terminal set consisted of components derived from existing LCHs, integer constants and variables such as: problem instance attributes. For example, the due date, current time, min and max due date and queue time. The GCHs are shown to perform equal or better than known construction heuristics. The performance of the GCHs was competitive to an algorithm called Johnson's algorithm which was the benchmark for the problem domain.

Jakobovic et al. [184] used genetic programming to create GCHs for a multiple machine scheduling problem. A steady state control model was used. Standard crossover is used and three types of mutation are used, namely, subtree mutation, permutation and shrink mutation. The functions used are arithmetic operators and the ramp function. The terminal set consisted of: processing time of a job, sum of processing times of remaining jobs and setup time from previous job. These are similar to those used in [183]. One hundred and twenty problem instances were used for a training set. The GCH performed better than already existing

heuristics such as earliest due date, shortest processing time and Montagne [182] on problem instances that were not part of the training set.

Ho and Tay [185], [186] used genetic programming to create GCHs for the job shop scheduling problem. The function set included arithmetic functions and an automatically defined function to encapsulate a subtree for reuse. The components used were problem instance variables for example; release date, processing time, number of operations for each job and average total processing time. A training set of 108 job shop scheduling problem instances was used. It was found that the GCHs were better than the best identified LCH of estimated due date.

Hyde et al. [11] used genetic programming to create GCHs for the bin packing problem. The function set consisted of arithmetic operators. The terminal set consisted of: features of the problem e.g. number of pieces in a bin, bin capacity and size of current piece. The best GCH found matches the performance of the first fit heuristic. It was found that the training set needs to represent the characteristics of the problem instances that need solving or the created heuristic may violate hard constraints. GCHs for one category of a problem, perform better when applied to problems from the same category. The GCHs can be specialized on a particular variation of a problem or be a general heuristic. It was found that low-level heuristics that were evolved to be more generic to the problem domain performed poorly overall. The work by Hyde et al. [11], [149] was extended for the bin packing problem [187]. In this work construction heuristics are evolved over a number of different problem instances. In total 20 instances were used as a training set. The evolved heuristics are applied to problems that differ to those in the training set. The GCHs, performed better than the best fit heuristic.

Burke et al. [188] used genetic programming to create GCHs for the two-dimensional stock cutting problem. The function set consisted of arithmetic operators. The components used were problem specific variables such as the width or height of a piece, the area of a piece or the slot height or sheet height. One GCH is evolved for instance. The GCHs are equivalent to the best fit heuristic. The GCHs were reported to have higher runtimes than the best fit heuristic.

Drake et al. [135] used genetic programming to create GCHs for a knapsack problem. These heuristics were trained on 5 instances and tested on an unseen set of 5 instances. The GCHs had equivalent performance to the existing LCH called profit to weight ratio. LCHs were evolved using genetic programming by Allen et al. [189] for the three dimensional knapsack packing problem. The components used were problem specific features such as the volume of a piece, the value of a piece and the coordinates of the current corner. The GCH is used to select from all available pieces, pieces are inserted one at a time. The evolved LCHs outperform a human created heuristic [190]. The performance of the GCHs were found to perform better than a simulated annealing meta-heuristic on 8 of 20 instances, two of which had a difference of 14.3% and 20.5% in the quality of the solutions compared to simulated annealing. The simulated annealing meta-heuristic had better performance for 10 of 20 instances but the largest difference in the quality of solutions was 11.5%.

Xie et al. [191] used genetic programming to create GCHs for the storage location assignment problem. Arithmetic functions were used and so were two custom functions. The first was an if statement, that returns one of two values based on the value of a third input. The second was a less than or equal to operator that returns 1 or -1. The components used were problem specific variables such as, the average picking frequency, the picking frequency of the most and least popular items, the sum of the total picking frequencies and the number of available and used bins. The best GCH found, had good reusability, was able to find near optimal solutions and had good runtime performance, when applied to increasingly challenging problem instances. The best GCH found was compared to the branch and cut method. The branch and cut method found solutions of better quality compared to the GCH. The hyper-heuristic had better scalability and

was able to solve problems of increased problem size which were computationally infeasible for the branch and cut method. An analysis of human created heuristics was given showing superior performance and flexibility of automatically created GCHs. It was shown that it is difficult to manually create an effective heuristic.

Pillay and Banzhaf [176] created hierarchical combinations of graph colouring LCHs. These combinations were investigated for the uncapacitated examination timetabling problem. In this approach a tree like hierarchy was made and the heuristics are applied using a pareto comparison instead of sequentially. The combinations are tested on the Carter benchmarks [158]. This preliminary work performed well producing solutions close to the best results cited in literature. This is made more impressive as no improvement phase was utilized. The results were generally better than tabu search [127], [192], variable neighbourhood search [193] and a fuzzy logic expert system [194].

Generative constructive hyper-heuristics is still a developing field. It was said in [189] that the first aim should be to automate the generation of LCHs and not necessarily obtain the best results. There is still potential for improvement and eventually replacing human experts in LCH creation. The human researcher will still be needed to determine training sets and components. Evolutionary algorithms have been shown to be good at combining components. For reusable GCHs a training set needs to feature all variations of the problem instances to achieve a high generality. Disposable evolved LCHs offer better performance generally for the problem instances they were evolved for. This means that the generality of the GCH is often sacrificed for improved results on specific instances. GCHs are already competitive with human designed construction heuristics.

5.5 Generative perturbative hyper-heuristics

The generative perturbative hyper-heuristic model makes use of existing features of the problem, LPHs, heuristics and algorithms. These are combined in some program structure to generate new perturbation heuristics. These components are specific to the problem domain [130], [151]. Generative perturbative hyper-heuristics create generated perturbation heuristics (GPH).

Fukunaga [130] used a population based approach that generates s-expression individuals called CLASS. This is done using a single operator called, composition. The composition operator combines the two individuals into a single s-expression using a conditional if-else statement, where the condition is a Boolean expression. The aim of the approach is to create variable selection heuristics for the SAT problem. This is considered early work in automatically creating perturbation heuristics. The work combined known components. These components were derived by looking at WalkSAT, GSAT and Novelty+ algorithms. The primitives for the approach were inputs of scoring of variables, ranking of variables, the age of the variable and conditional branching functions. Two evolved perturbative heuristics were compared with WalkSAT and Novelty+ algorithm and found to be competitive. CLASS was shown to be better than random generation of GPHs. In [131] the work by Fukunaga was extended and the GPHs were found to have good scalability. In [195] the function set is increased. The GPHs are shown to have competitive performance with respect to the previous version [130] and GSAT and Novelty+ algorithms.

Nguyen et al. [196] used genetic programming to evolve disposable heuristics. This work was done using the Hyflex framework. This study is an exception to the use of Hyflex as a selection perturbative hyper-heuristic framework. In this study it uses the LPHs within Hyflex as part of a new evolved heuristic. The function set consists of conditionals, acceptance methods and

combination functions. Conditionals included: if new solution, if local optimum and if no improvement. Acceptance methods used were: accept all moves, simulated annealing. A combination function was available to combine two LPHs, executing both heuristics sequentially. It was found that the evolved disposable heuristics performed better than heuristics generated randomly from the initial population. On the bin packing and SAT problems, the generated heuristics were competitive to the existing hyper-heuristics provided by CHeSC2011. The performance of existing hyper-heuristics was not competitive with competition submissions for CHeSC2011.

Poli and Keller used linear genetic programming to evolve perturbation heuristics [197]–[201]. This approach evolved perturbation heuristics for the data sets from TSPLIB [179] for the travelling salesman problem. The terminal set is made up of local search heuristic variants of 3-opt, 2-opt and includes a basic hill climber. The terminal set also includes if_2-Change and if_3-Change, which execute 2-opt and 3-opt respectively only if an improvement is made to the tour. The function set includes If_improvement, which branches if an improvement occurs and repeat until improvement, which repeats until an improvement is made. The results of [200] were shown to be better than those obtained by Oltean [177] which were obtained using a GCH created by a generative constructive hyper-heuristic. In [197] it was found that reducing the number of primitives used effects the time it takes to evolve heuristics as the search space is reduced. The results of [200] were similar in quality to a genetic algorithm solver for the travelling salesman problem.

Bader-el-den and Poli [202] used genetic programming, using strong typing in the form of a grammar to create disposable perturbation heuristics to solve the SAT problem. This was done by using a number of already known local search heuristics. A grammar was created using features found in the local search heuristics. One feature identified was probabilistic branching. This approach used probabilistic branching components which were if statements with a probability variable as an argument. This was used instead of conditional branching for program structure. The initial population was reported to contain valid heuristics. Conditional branching was considered a weakness of the heuristics generated by Fukunaga [130]. This was because it did not give the GP system enough freedom and was used to choose between two heuristics in the conditional which resulted in slow runtimes of the GPHs created. The GPHs in this study were found to produce results competitive with the WalkSat algorithm and the FlipGA algorithm in [203].

Drake et al. [132] used grammatical evolution to create an algorithm for the shaking method for a variable neighbourhood search and construction heuristics for the vehicle routing problem. The approach was tested on two different categories of vehicle routing problems. It performed better on capacitated vehicle routing problems (where there are more constraints on the vehicles) than on vehicle routing problems with a time window (where an added constraint is included to deliver within a certain time period). This could be due to the components made available to the hyper-heuristic not being sufficient to deal with the constraints of the second type of problem. This suggests that the provided functions and terminals were not able to form heuristics capable of dealing with differences in the problem domain.

Burke et al. [134] used grammatical evolution to evolve perturbation heuristics for bin packing problems. A single instance from each of the selected categories of bin-packing problem instances was used to evolve GPHs. During the evolution the GPHs were measured by their performance over 100 applications of the same GPH on a different initial candidate solution for each generation. This hyper-heuristic was able to generate heuristics that were one away from the global optimum for the tested bin packing problems. The advantage of using a generative perturbative hyper-heuristic to automate the design of LPHs is that GPHs can be evolved specifically for any bin packing problem.

Sabar et al. [204] used grammatical evolution to evolve combinations of LPHs and acceptance methods. This approach used a steady-state control model. In genetic programming terms the LPHs and acceptance methods are part of the terminal set. The function set consisted of: neighbourhood union, random gradient and token ring search. Neighbourhood union combines LPHs. Random gradient involves repeated application of a heuristic until it is not accepted by the acceptance methods. This approach aims to combine acceptance methods and existing LPHs. In total 10 acceptance methods were used. The approach was tested on examination timetabling and vehicle routing problems. It produced competitive results compared to bespoke local search methods such as simulated annealing and large neighbourhood search.

KhudaBukhsh [195] used local search to generate variable selection heuristics for the SAT problem. A GPH was created by performing a local search of components identified by Fukunaga [130]. The generated heuristics can be considered superior to those evolved by Fukunaga due to ability to solve challenging SAT instances and having produced results that were competitive with state of the art SAT solvers. This hyper-heuristic created heuristics that were better than the GNOV, RANOV and AE20 heuristics.

There is little work done on this type of hyper-heuristic. The work by Sabar et al. [204] in exploring the combination of LPHs and move acceptance methods is promising. Adding operators to add conditional branching or problem specific conditionals could be explored to create more effective perturbative heuristics. The fitness function should be tailored to the aim of the approach as in the work by Fukunaga [133] and Burke et al. [131]. LPH combinations are disposable in most optimization problems due to the problem specific nature of LPHs.

5.6 Critical analysis of hyper-heuristics

The most researched type of hyper-heuristic is the selection perturbative hyper-heuristic. The performance of selection perturbative hyper-heuristics is dependent on the ability of heuristic selection and move acceptance methods. For single-point search algorithms two distinct methods are usually required. For multi-point search algorithms these are one and the same. Provided that the move acceptance method is good the heuristic selection method can be effective even if it is very simple, even simple random selection has been shown to work well. The studies surveyed in the chapter show that meta-heuristics such as tabu search and simulated annealing have performed well as move acceptance methods. It has been shown that simulated annealing is a top performing move acceptance method. For the project presentation problem an ant algorithm hyper-heuristic and the guided operator genetic algorithm hyper-heuristic outperformed a choice function selection hyper-heuristic. This suggests that biologically inspired methods may have advantages in certain problem domains. It is evident from the literature that genetic algorithm hyper-heuristics showed promise..

Selection constructive hyper-heuristics are good at creating high quality initial solutions if LCHs are available for the domain. It is interesting to note how combinations of heuristics can be applied to create candidate solutions. There is merit in searching both the low-level heuristic space and the solution space. When employed as selection constructive hyper-heuristics multi-point searches have been shown to be quite effective. Genetic algorithms are seen to produce good results for a number of domains.

Generative constructive hyper-heuristics show that there is potential in automating the creation of new LCHs. It is clear that the components made available to the approach must be carefully considered to be able to represent the problem for which a GCH is being evolved. The generality of these GCHs can be improved by featuring a larger training set but this would increase the time and reduce the efficiency and efficacy of the GCH. The effort of evolving

more general low-level heuristics is generally worth the cost of evolving essentially disposable low-level heuristics for a domain. Evolutionary algorithms such as genetic programming and grammatical evolution have been found to generally be suited for the purpose of generative constructive hyper-heuristics.

Generative perturbative hyper-heuristics offer a way to produce new perturbative heuristic combinations. The aim is to produce a good performing LPH. Generative perturbative hyper-heuristics could also be used to identify strong components, such as good move acceptance and or good combinations of LPHs. Genetic programming has been very effective at producing low-level heuristics both construction and perturbation.

5.7 Summary

This chapter presents a literature survey on hyper-heuristics. Firstly hyper-heuristics are introduced, then the four types of hyper-heuristics are presented: Selection perturbative, selection construction, generative constructive and generative perturbative. Finally a critical analysis of hyper-heuristics is given.

Chapter 6 Nurse rostering using Hyper-Heuristics

This chapter looks at hyper-heuristic approaches used to solve the nurse rostering problem. It also attempts to categorize LPHs for the nurse rostering problem.

6.1 Nurse Rostering and selection perturbative hyper-heuristics

This section looks at the state of the art hyper-heuristic approaches for solving the nurse rostering problem. Hyper-heuristics that have been used to solve the nurse rostering problem have essentially been selection perturbative hyper-heuristics.

Cowling et al. [205] used a selection perturbative hyper-heuristic to solve a nurse rostering problem for a UK hospital. A choice function was used to select LPHs. No move acceptance method is described. The LPHs used in this study are based on neighbourhood operators used in Dowsland [206]. The LPHs used in this study will be referred to as the Cowling heuristics, these were:

- H1 – Randomly select a nurse, randomly select a week, randomly change the shift types in that week for the selected nurse.
- H2 – Same as H1 but only accepts moves that result in an improvement in hard constraint violations.
- H3 – Same as H1 but only accepts moves that result in an improvement in hard constraint violations and no worsening soft constraint violations.
- H4 – Same as H1 but only accepts moves that result in an improvement in soft constraint violations.
- H5 – Same as H1 but only accepts moves that result in an improvement in soft constraint violations and no increase in hard constraint violations.
- H6 – Randomly select a nurse, if the nurse's roster violates the coverage constraint(section 4.1), change all shift types to the opposite shift type. e.g. a day shift becomes a night shift.
- H7 – Same as H6 if the resulting move still violates the coverage constraint, randomly select another nurse working the opposite shift type and swap rosters.
- H8 – Same as H1 but the replaced weekly shifts are stored. It is attempted to replace a week of the roster of another randomly selected nurse with the stored weekly shifts of the first nurse. This is attempted until a move is found that improves the soft constraint violations.
- H9 – Same as H8 but accepts moves that result in an improvement in soft constraint violations with no hard constraint violations.

This hyper-heuristic was applied to 52 instances of the nurse rostering problem specific to a UK hospital. 27 of the instances were within 10% of the optimal solution and 9 were solved consistently to optimality. This approach was considered more reliable than a genetic algorithm as it was able to solve more instances consistently. This hyper-heuristic produced more reliable results than genetic algorithms and was able to produce a feasible solution for an instance which the genetic algorithms [207] could not. It was considered equivalent to a tabu search algorithm [206] in terms of producing feasible solutions. The hyper-heuristic approach generally had

worse constraint violations than the bespoke methods it was compared to.

Burke et al. [208] used a random selection perturbative hyper-heuristic with a tabu list to rank the heuristics used; this allows selection of the best performing LPHs. The LPHs used were the Cowling heuristics. The hyper-heuristic was applied to the same 52 instances used by Cowling et al. [205]. This hyper-heuristic performed better than the choice function hyper-heuristic used by Cowling et al. [205].

Bai et al. [209] used a selection perturbative hyper-heuristic that was hybridized with a genetic algorithm. The LPHs used were the Cowling heuristics. The selection of LPHs was done using a choice function. Simulated annealing was used to decide whether or not to accept the move made by the chosen LPHs. The hyper-heuristic is employed after the genetic operators are applied to the candidate solution. This hyper-heuristic performed better than the tabu search hyper-heuristic by Burke et al. [208] on 22 of the 52 instances. This result was shown to be statistically significant.

Bilgin et al. [210] use a selection perturbative hyper-heuristic to solve Belgian nurse rostering problem. This hyper-heuristic used random selection to select heuristics and simulated annealing move acceptance. In the problem model used a shift assignment consists of a shift type, skill type, day and nurse quadruple. This means that a shift assignment can be made without a compatible skill type for the given shift type. A comparison with great deluge as the move acceptance method was made. The LPHs used were based on the neighbourhood moves described in [211]:

- Assign shift – Allocate a randomly selected shift type, to a randomly selected nurse on a randomly selected day. The skill type is given by the skill associated with the given shift type.
- Delete shift – Randomly select a nurse, randomly select a shift assignment and remove the shift type and skill type.
- Single shift day – Randomly select a nurse, randomly select a working day from that nurse's roster, randomly select another nurse who does not work on the same day and has the same skill type. Move the shift assignment to the nurse who was not working on that day.
- General assignment change – Randomly select a nurse, randomly select a shift assignment from the nurse's roster. Randomly change the shift type to another shift type.
- Change assignment based on compatible shift type – Randomly select a nurse, randomly select a shift assignment from the nurse's roster. Randomly change the shift type of the shift assignment to a different shift type that fulfills the coverage constraint for the selected day of the shift assignment.
- Change assignment based on compatible skill type – Randomly select a nurse of those with more than one skill type, randomly select a shift assignment from the nurse's roster. Randomly change the shift type of the shift assignment to a different shift type that has the same skill type.

It was found that the hyper-heuristic with simulated annealing move acceptance performed best and was better than the variable neighbourhood search used in [212].

Bilgin et al. [109] used a random selection hyper-heuristic for the INRC2010 competition. LPHs are selected randomly and the moves are accepted through the simulated annealing move acceptance method. Following the application of the low-level heuristic to a candidate solution, the candidate solution is changed using a greedy shuffle algorithm. Bilgin et al. used set of a dozen LPHs. These were derived from three categories:

- Swap subset of day shifts – Randomly selects a subset of all days for two randomly selected nurses and perform a swap of their corresponding shift assignments.

- Swap subset of weekend shifts – Randomly selects a subset of all weekend shift assignments for two randomly selected nurses and perform a swap of the corresponding shift assignments.
- Swap subset of weekday shifts – Randomly selects a subset of working days for two randomly selected nurses and perform a swap of the corresponding shift assignments.

This hyper-heuristic was placed third in the competition for the long instance track, the problems of which were seen as more challenging in the competition. This suggests that this hyper-heuristic was good at dealing with the more difficult instances and was better suited to time consuming or challenging problems. The difference between this work and most other selection perturbative hyper-heuristics was the additional use of a greedy shuffle algorithm after the hyper-heuristic for 20% of the remaining time allocated to each problem instance. The greedy shuffle algorithm considers incremental swaps of shift types between all nurses and all days in the scheduling period which results in improving or equal soft constraint violations. If no improvement was made by greedy shuffle for a number of iterations, the schedule is changed by swapping the nurse's roster with the highest soft constraint violations with the roster of a randomly selected nurse. Similar greedy shuffle algorithms have been found to produce improved solutions for the nurse rostering problem [104], [119], [120].

Bilgin et al. [213] studied selection perturbative hyper-heuristics for the nurse rostering problem. This study explored a variety of move acceptance (great deluge, simulated annealing, and accept improving or equal moves) and heuristic selection methods (simple random, choice function and a dynamic heuristic set strategy with simple random) in [213]. The LPHs used were the same as those in [109] as described above. The most effective hyper-heuristic was found to be a random selection with simulated annealing for move acceptance. These hyper-heuristics were tested on the INRC2010 benchmark instances.

Anwar et al. [214] used harmony search as a hyper-heuristic. Instead of changing variables for the problem domain in a harmony memory it instead selects LPHs. A memory to store candidate solutions and a memory to store selected LPHs was used. New solutions are accepted if an improvement is found over a randomly selected candidate solution in memory. New solutions that are accepted replace previous solutions stored in memory. This hyper-heuristic used 4 LPHs:

- Swap two shifts – A nurse with a working shift type is randomly selected and swapped with a working shift type of another randomly selected nurse for the same day. The shift types are not allowed to be the same.
- Move one shift – A nurse is randomly selected and a random shift assignment is selected and swapped with another nurse's shift type on the same day. The shift type can be any type, even not worked.
- Shuffle moves – Selects the nurse with the roster with the highest soft constraint violations and attempts to incrementally swap a subset of the shift assignments (worked or not worked) from a randomly selected day until the end of the scheduling period with a randomly selected nurse.
- Blank move – No changes are made to the candidate solution

This hyper-heuristic was tested on the INRC2010 benchmark data set but only on the early instances for the sprint, medium and long tracks. For the sprint instances it was able to only achieve the best known score for 2 instances (sprint_early05 and sprint_early08), Runtimes were over 30 minutes. It is understandable that population (multi-point searches) based hyper-heuristics require more runtime than single-point searches. These results are possibly due to the LPH set being too small and or not being specific enough to the problem domain to exploit the solution space.

6.1.1 Hyflex

This section looks at how entries into the CHeSC2011 [215] competition solved the nurse rostering problem using the Hyflex framework [141]. The focus of the following studies is to look at how well they performed on the nurse rostering problem and the competition in general.

Misir and De Causmaecker [216] developed a hyper-heuristic for CHeSC2011 competition. The LPH selection method is a probabilistic method. This probabilistic method calculates a performance metric to measure the effect of each LPH on the candidate solution. This method orders the LPHs by the improvement in the candidate solution quality. A reward-penalty strategy is used in this hyper-heuristic is used to adapt the parameters of the methods used during the execution of the hyper-heuristic. The move acceptance method used was adaptive iteration limited list-based threshold acceptance (AILLA). AILLA keeps a fixed list of previous best candidate solutions. A change to the candidate solution is accepted if it is better than the candidate solution. A change to the candidate solution is also accepted if after a number of iterations with no improvement the change is accepted if it is better than or equal to the next candidate solution in the list. This hyper-heuristic was the overall winner of the CHeSC 2011 competition. This hyper-heuristic was the 10th best hyper-heuristic for the nurse rostering track.

Chan et al. [136] created a hyper-heuristic based on the analogy of pearl hunting, this used repeated diversification and intensification stages. Offline learning in the form of generating a decision tree decides which LPHs to apply during intensification. The intensification stage uses two methods. The first method executes local search LPHs with a low search depth until an improvement is found and stores a small number of the best solutions. The second method executes hill climbing LPHs with a high search depth until no improvement can be found. The second method attempts to improve only the best solutions found in the first method. During the intensification stage only improving moves are accepted. The diversification stage uses one randomly selected LPH that is not a local search low-level perturbation heuristic. Diversification moves are accepted if they meet a threshold set by the best result of the first iteration of the intensification stage. This approach found 3 new best known results for the nurse rostering problem benchmark instances. This approach achieved fourth place in the CHeSC2011 competition and was placed sixth for nurse rostering.

Hsiao [217] used a variable neighbourhood search (VNS) as a hyper-heuristic for the CHeSC2011 competition. The shaking step of VNS selected the mutation and ruin and recreate LPHs and used a tabu list to limit the use of poorly performing LPHs. The VNS hyper-heuristic then iterated through the use of the local search heuristics for the local search step, where local search LPHs are selected using rank selection. A population of solutions is kept. Tournament selection is used as a move acceptance method. This hyper-heuristic achieved second place overall in the competition. This approach scored best compared to other entries in the CHeSC2011 competition for the nurse rostering problem.

Larose [218] developed a hyper-heuristic called MLAlgorithm to compete in CHeSC2011. This algorithm combines iterated local search with reinforcement learning. The heuristic selection was a modified reinforcement learning algorithm from the work by Meignan et al. [219]. This approach consists of three stages, diversification, intensification and move acceptance. The move acceptance accepts an improving candidate solution or a candidate solution after no improvement has been made for a number of iterations. For the diversification stage the ruin and recreate and mutation LPHs were applied to the candidate solution, there was a chance that no change would be made to the candidate solution. The intention of allowing a possible no change to the candidate solution was intended so the intensification stage could further attempt to improve a candidate solution. The intensification stage used local search LPHs to improve the candidate solution until the solution stops improving. MLAlgorithm achieved third place overall

in the competition. MLAlgorithm achieved the second best score for the nurse rostering problem compared to other entries in the CHeSC2011 competition.

Lehrbaum [220] created a hyper-heuristic called HAHA for the CHeSC2011 competition. HAHA consists of three main phases: serial search, 'generate mutations' and parallel search. The phases are executed in sequential order until a time limit is reached. The move acceptance used during all three phases was improving or equal acceptance but there is a limited list of already visited solutions which are rejected if revisited. If solutions found by the stages are all in this list than the best candidate solution is changed using mutation LPHs until it is not present in the memory. Serial search is a stage that uses only local search LPHs. Serial search applies all local search LPHs in order of their measured performance to the candidate solution. If there is an improvement the move is accepted and a random search depth is set. Generate mutations is a stage that uses mutation LPHs. The mutation LPHs are selected using roulette wheel selection and applied seven times to the candidate solution. The seven applications create seven different candidate solutions for the parallel search stage. Parallel search applies the sorted list of LPHs across the seven available candidate solutions. The LPH used will change if there is no improvement after a number of iterations. This stage attempts to improve the solution by using LPHs across different candidate solutions. If no improvement to the current best candidate solution was found in the parallel search stage, a solution is selected from the list using roulette wheel selection. If the selected solution has already been visited, mutation LPHs are applied until the candidate solution has changed. This hyper-heuristic scored sixth overall for CHeSC2011 and third with respect to the nurse rostering problem.

From these studies it can be seen that a selection perturbative hyper-heuristic such as that developed by Misir and De Causmaecker [216] which generalized the best over multiple problem domains, did not perform well on the nurse rostering problem domain. This suggests that some hyper-heuristics will perform better for different problem domains. In this dissertation the performance of perturbative hyper-heuristics is investigated for the nurse rostering problem. It can also be seen that methods that were employed by selection perturbative hyper-heuristics which used mechanisms from multipoint search, were prevalent in these studies and performed well specifically for the nurse rostering problem domain.

6.2 Categorization of low-level heuristics for nurse rostering

This section attempts to categorize the LPHs used in hyper-heuristic studies and neighbourhood operators used by non-hyper-heuristic studies for the nurse rostering problem. The reason for looking at non-hyper-heuristic studies is that many selection perturbative hyper-heuristics look to previous work using search and neighbourhood operators when deciding upon low-level heuristics, often neighbourhood operators are used as the LPHs in hyper-heuristic studies. For example the neighbourhood operators used by Dowsland [206] were adapted for selection perturbative hyper-heuristics by Cowling et al. [205], Burke et al. [208] and Bai et al. [209]. The neighbourhood operators used by Bilgin et al. [211] were also used in a selection perturbative hyper-heuristic for nurse rostering [210].

The LPHs currently used for nurse rostering can be divided into two categories. The first category will be referred to as swap heuristics and the second category will be referred to as edit heuristics for the purposes of categorizing nurse rostering low-level perturbation heuristics. Swap heuristics involve exchanges of shift assignments which are already present in a given schedule. The swap heuristic category is subdivided into five sub-categories, namely: Swap two

shifts (s1), swap a shift with a free shift (s2), swap *n* shifts (s3), swap using problem specific conditions (s4) and swap with move acceptance (s5). Swap heuristics of s1-s3 may fall into the categories of s4 and s5. Swap heuristics could easily be referred to as swap heuristics however using the term local differentiates them from the counterpart of heuristics that perform changes to a given schedule. Edit heuristics involve adding, removing or changing shift assignments. Edit heuristics work by performing new changes to the given schedule. Edit heuristics are generally used to solve nurse rostering problem instances which have the coverage constraint as a soft constraint instead of a hard constraint. The edit heuristic category is subdivided into five different sub-categories, namely: add and remove (e1), change shift type (e2), change *n* shifts (e3), change using problem specific conditions (e4) and change with move acceptance (e5). Edit heuristics of e1-e3 may fall into the categories of e4 and e5.

The categorization scheme will be presented in the format: sub-category 1 (sub-category *x*, sub-category *y* ...). This defines that the first sub-category outside the parenthesis, is the main parent sub-category of the LPH. Sub-categories within the parenthesis are those which have additional requirements in addition to the parent sub-category. For example, a swap of multiple shifts between two nurses' rosters with the same contract would be categorized as s3(s4) but if this LPH were to also require that the change improves the soft constraint violations then it would be categorized as s3(s4, s5). When there are no parentheses additional sub-categories only fall into a single sub-category. For example, swapping a free shift of one nurse with a worked shift of another is simple categorized as, s2. When a second sub-category is identified a + sign is used to denote that the category is also used. e.g. sub-category 1 + sub-category 2(sub-category *x*, sub-category *y*). For example if a LPH existed that performed first a swap of two shifts between nurses but then only performed a swap of multiple shifts between the same nurses if the soft constraint score was reduced by the change it would be categorized as, s1 + s3(s5). A LPH for nurse rostering can fall into sub-categories from both the local and global categories.

6.2.1 Swap heuristics

Swap heuristics in nurse rostering are an exchange of one shift assignment from one nurse to another. Swap heuristics have to select and swap from existing assignments in the given schedule. There are five sub-categories of swap low-level heuristics found in hyper-heuristic studies for solving the nurse rostering problem. These swap heuristic sub-categories are: Swap two shifts (s1), swap a shift with a free shift (s2), swap *n* shifts (s3), swap using problem specific conditions (s4) and swap with move acceptance (s5). More than one sub-category can form the categorization of a swap heuristic.

The following subsections describe each of the 5 swap heuristic sub-categories (s1-s5) and their use in hyper-heuristic and non-hyper-heuristic studies for use in solving the nurse rostering problem.

6.2.1.1 Swap two shifts (s1)

This swap heuristic is an exchange of any two selected shift types from two different nurses' rosters on the same selected day and can include or exclude swapping free shifts. LPHs from in the study by Anwar et al. [214] fall into this sub-category.

The neighbourhood operators used in the non-hyper-heuristic by Lü and Hao [106] also fall into this sub-category. It was specified for this neighborhood operator that both shift assignments must not be the same shift type or a free shift. This is necessary as it ignores unnecessary swaps as exchanging the same shift type would have no effect on the given schedule. It is possible to include free shifts but this would be categorized as both an s1 and s2 swap heuristic. Generally this LPH is forbidden from swapping shift types that are the same e.g. swapping the shift types

of two nurses working morning shifts on the same Monday would not improve the roster of either nurse.

6.2.1.2 Swap a shift type with a free shift (s2)

This swap heuristic is applied to a nurse with a shift type and another nurse with a free shift on the same day. An LPH of this sub-category is found in the study by Bilgin et al. [210] for hyper-heuristics for nurse rostering.

Generally, this sub-category of swap heuristics can be identified most often in non-hyper-heuristic studies. The heuristics in this sub-category explore a smaller neighbourhood compared to s1. Neighbourhood operators in this sub-category were used in the study conducted by Lü and Hao [106] and this approach was effective at solving the nurse rostering problem. Neighbourhood operators in this sub-category can also be found in the studies conducted by: Frøyseth et al. [76], [105] and Burke et al. [104].

6.2.1.3 Swap *n* shifts (s3)

This sub-category of local heuristic performs a bigger change to the given schedule compared to a l1 local heuristic. It follows that it generally explores a larger search space than l1 and l2 local heuristics. There are multiple variants of this sub-category of local heuristic and some methods if used as low-level perturbation heuristics will fall into this sub-category e.g. greedy shuffle, which would be a l3(l5) local heuristic as it swaps *n* shifts but also accepts improving moves. The l3 sub-category is the most flexible local heuristic categorization as it can be defined in a variety of ways. Heuristics that fall in this can consider multiple days in sequence or randomly and can also perform exchanges between multiple nurses.

Cowling et al. [205], Burke et al. [208] and Bai et al. [209] used a heuristic referred to as H7 which swaps the roster of the recently changed nurse with another nurse working the opposite shift type (days or nights). This is the only LPH which falls into both swap and edit categories for nurse rostering.

Anwar et al. [214] used a swap heuristic called shuffle moves. This heuristic swaps blocks of shift assignments between nurses. It is stated that it considers both worked and not worked shifts and swaps a subset of shift assignments. Bilgin et al. [109], [213] used local heuristics which swapped subsets of shift assignments between nurses.

Frøyseth et al. [76], [105] used two neighbourhood operators that fall into this sub-categorization. The first swaps three shift assignments for three nurses on a single day. The second swaps two shifts for two nurses on two days.

Vu et al. [107] used two neighbourhood operators which swap two and three days between two nurses. Awadallah et al. [116], [118], [121], used a neighbourhood operator called 'swap three days'. This swaps a selected shift assignment and two consecutive shift assignments with a randomly selected nurse's corresponding shift assignments. This differs from other s3 LPHs, swapping contiguous shift assignments.

6.2.1.4 Swap using problem specific conditions (s4)

Swap heuristics that fall into any of the other three sub-categories of swap-heuristics fall into this sub-category when additional problem specific requirements are introduced. For example, selecting the nurse with the roster with the most soft constraint violations for an s1, s2 or s3 swap heuristic, or that a contract requirement such as skill type is a criterion for the nurse

chosen in the exchange of shift types for a selected day or days. These are the least flexible as they rely heavily on the constraints present in a specific problem instance. Heuristics falling into this sub-category will generally be identified as having a parent sub-category from the sub-categories of s1-s3. For example, a swap of two shifts between nurses which only occurs between nurses with the same contract would be categorized as s1(s4).

Bilgin et al. [109], [213] used LPHs which swap a subset of shift assignments between nurses based on the definition of a weekend (which is specific to the nurse's contract). That is these swaps used problem instance specific information to perform exchanges between nurse's rosters. In these problem instances a weekend could be defined as Saturday, Sunday or as Friday, Saturday and Sunday or as Saturday, Sunday and Monday. These LPHs would be categorized as s3(s4).

Anwar et al. [214] used a shuffle moves swap heuristic which selects two nurses, one is selected randomly and the other is the nurse with the roster that incurred the highest soft constraint violations. This heuristic iterates for each day in the scheduling period, the heuristic swaps an increasing number of shifts at each iteration between the two nurses. This is categorized as s3(s4).

Awadallah et al. [116], [118], [121], used a neighbourhood operator based on satisfying the 'complete weekend' and 'identical shift types during weekend' constraints. If the complete weekend constraint is violated the shift type is swapped with another nurse's shift type (an s1 LPH) and additionally a further swap is performed to make the shift types during the weekend identical. These would be categorized as s1(s4)

6.2.1.5 Swap with move acceptance (s5)

Heuristics in this sub-category have a criterion or method to determine whether the move made by a heuristic is accepted or rejected. Like the previous sub-category (s4) heuristics in this sub-category fall into the sub-categories of s1-s3 with a move acceptance criterion. Swap heuristics with this sub-category are found as neighbourhood operators. The neighbourhood operator 'selective partial swap' used by Tassopoulos et al. [108] falls into the categorization of s3(s5). The heuristic performs an iterative number of shift assignment swaps, over each day in the scheduling period between nurses. This falls into the s3 sub-category. At each swap a probability decides whether to accept or reject the move made and only improving or equal moves can be accepted. This means it falls into the s5 sub-category.

6.2.1.6 Summary of swap heuristic category

This categorization presents the three sub-categories that describe the basic exchanging of shift assignments between nurses as LPHs found in literature. The sub-categories of s4 and s5 describe additional requirements which are additional requirements of heuristics that fall into the sub-categories of s1, s2 or s3. It is possible to categorize some swap heuristics as being both s3 and s4. A swap heuristic in this category will exchange two working shifts or more with an additional requirement such as a skill required for working the shift assignment. If an LPH was used which performed an s1 swap and then did an s3 swap it would be categorized as s1+s3. The sub-categories of s4 and s5 are identified as any sub-category of s1, s2 and s3 which has the either a problem specific requirement (s4) or a requirement to accept or reject the move made by the swap (s5). The sub-category of s2 is not by default an s4 sub-category as the LPH is requesting a swap with a nurse that has a free shift on the selected day and this is not specific to a problem instance detail but rather a feature of the problem domain as a whole. A soft constraint violation of a nurse's roster would be specific to the problem instance as would the definition of a weekend, thus swap heuristics which make changes depending on these are

categorized as the s4 sub-category of swap heuristic.

The application of heuristics falling into the s1 sub-category result in a small change. The heuristics which fall into this category explore a small neighbourhood as only two shifts are swapped between nurses and they cannot target nurses that have been assigned free shifts. It is not possible generally for this category of LPH to be able to fully solve the nurse rostering problem. Therefore it is necessary to search an additional neighbourhood that of exchanging worked shifts and free shifts, this has been categorized as the s2 sub-category. It can be seen from literature that heuristics that fall into the s1 and s2 sub-categories of LPH are used together.

The sub-category s3 includes any heuristic swap between two or more nurses which exchanges two or more shift types. This sub-category could be further sub-divided into LPHs that perform swaps between more than two nurses. This is unnecessary as the function is the same, a number of shift types are exchanged in either case. This further sub-division could be seen as another sub-category. The LPHs falling into this sub-category of swap heuristics, operate in a large search space that encompasses the neighbourhoods of the sub-categories s1 and s2. These LPHs can result in large changes to the solution space and convergence to a local optima may be avoided.

The sub-category s4 is useful in that it focuses the search space of LPHs in the sub-categories of s1-s3. This sub-category is useful in that it can utilize problem specific information to better traverse the search space. There is a risk however that if the problem specific information is no longer advancing the search then LPHs with this sub-category can be redundant.

The sub-category of s5 seeks to limit the search space by only applying swaps that are seen as beneficial to the search. This could be seen as lessening the work of an acceptance method as these LPHs would exclude changes to the solution space which are worsening.

Table 6.1 Hyper-heuristics and swap heuristic categorization of LPHs used

Hyper-heuristics	Swap heuristic sub-categories
Anwar et al. [214]	s1, s3
Bilgin et al. [210]	s2
Bilgin et al. [109] [213]	s3, s3(s4)
Cowling et al. [205]	s3
Burke et al. [208]	s3
Bai et al. [209]	s3

Table 6.2 Non-hyper-heuristics and swap heuristic categorization of LPHs used

Non-hyper-heuristics	Swap heuristic sub-categories
Awadallah et al. [116], [118], [121],	s1, s3, s1(s4), s3(s4)
Bilgin et al. [211]	s2
Frøyseth et al. [76], [105]	s1, s2, s3
Lü and Hao [106]	s1, s2
Tassopoulos et al. [108]	s3(s4)
Vu et al. [107]	s2, s3, s3(s4)

Table 6.1 gives a summary of the hyper-heuristic studies for nurse rostering and the swap heuristic sub-categories found within those studies. Table 6.2 gives a summary of non-hyper-heuristic studies for nurse rostering and the swap heuristic sub-categories of neighbourhood operators used in these studies.

6.2.2 Edit heuristics

This category of LPH performs any possible change to a schedule but do not use the existing schedule's information. In some cases this is simply changing a shift type randomly in other

cases a successive add and remove of shift types, or an LPH that attempts to correct the occurrence of too many or too few shift assignments by the inserting or removing of too many shift assignments to the schedule. In this respect the change is not local to the given schedule. There are five sub-categories identified for this category namely, add and remove (e1), change shift type (e2), change *n* shifts (e3), change using problem specific conditions (e4) and change with move acceptance (e5). An edit heuristic can fall into more than one sub-category.

The following subsections will detail each of the three edit heuristic sub-categories (e1-e5) and their use in hyper-heuristic studies for solving the nurse rostering problem.

6.2.2.1 Add and remove (e1)

This sub-category of edit heuristic adds or removes shift assignments to a selected nurse's roster. Similar LPHs are used in other domains for example in the study by Raghavjee [30] for solving school timetabling problems heuristics for allocation and de-allocation of tuples were used. One of the consequences of this sub-category of edit heuristic is that too many shift assignments may be made or removed from the schedule. This could increase the time taken by the search as extra LPHs may be needed to remove excess shift assignments or to add necessary shift assignments.

Bilgin et al. [210] used both an assign shift and delete shift heuristic in a selection perturbative hyper-heuristic. These added and removed randomly selected shift assignments from randomly selected nurses.

6.2.2.2 Change shift type (e2)

This sub-category of edit heuristic contains heuristics which make a single change to the shift type of a randomly selected shift assignment of a randomly selected nurse.

Bilgin et al. [211] used a change shift LPH for a selection perturbative hyper-heuristic. The LPH would randomly change the shift type of a randomly selected shift assignment in a nurse's roster.

6.2.2.3 Change *n* shifts (e3)

This sub-category of edit heuristics contains heuristics which make changes to a number of selected shift assignments in a nurse's roster. The Cowling heuristics (see: Chapter 6 section 6.1) used by Cowling et al. [205], Burke et al. [208] and Bai et al. [209] all fall into this sub-categorization as they change a number of shift assignments. The LPH referred to as H1 performs random changes to shift types present in a selected week of a nurse's roster. H1 is used as the basis for the LPHs of H2-H5 and H8 and H9. The LPHs referred to as H6 and H7 change a nurse's entire roster. This is a large amount of change to be made and the application of the heuristic can move to a different area of the search space.

6.2.2.4 Change using problem specific conditions (e4)

This sub-category of edit heuristics performs changes using problem instance specific details. For example the skill type required to work a shift must be possessed by the selected nurse. This limits the number of changes that can be made by these heuristics and the areas of the search space they can explore. As with sub-categories s4 and s5 in the swap heuristic sub-categories, e4 is categorized as having a parent sub-category of e1, e2 or e3.

Cowling et al. [205], Burke et al. [208] and Bai et al. [209] used edit heuristics which fall into this sub-category. These LPHs target the shift types that violate the coverage constraint for a

randomly selected nurse (H6, H7). These heuristics also fall into the e3 sub-category of edit heuristics. This categorizes H6 as e3(e4) while H7 has an additional swap heuristic sub-category making its categorization e3(e4)+s3.

Bilgin et al. [211] used two edit heuristics of this sub-category which perform changes depending on the skill type required for shift types. The first edit heuristic will change a shift type to a different shift type (which has the same skill type) for a nurse that fits the required coverage constraint requirements for the selected day. The second edit heuristic will only select from a list of nurses which have multiple skill types, it changes a shift type to another shift type. In this case the problem specific condition is that only nurses with more than one skill type can be selected by this edit heuristic.

6.2.2.5 Change with move acceptance (e5)

This sub-category of edit heuristics includes heuristics with an additional requirement of a move acceptance criterion. If the move acceptance criterion is not met, the changes made are rejected. For example, a secondary requirement of an edit heuristic may be ‘accept only improving soft constraint violations’ to the move made by the heuristic. Edit heuristics categorized as e5 will have a parent sub-category from e1, e2 or e3.

The Cowling heuristics used by Cowling et al. [205], Burke et al. [208] and Bai et al. [209] had six of nine LPHs (H2-H6 and H8 and H9) which used a move acceptance criterion. The acceptance criteria used in these LPHs, was that there must be a no change or a reduction in hard constraint violations and/or soft constraint violations. This sub-category does limit the possible moves which can be made by heuristics in this sub-category.

6.2.2.6 Summary of edit heuristic category

Edit heuristics are LPHs which perform changes to the given schedule which are not already present in the schedule. The majority of the Cowling heuristics fit into the categorization of e3(e5). Application of these heuristics can focus the search. These edit heuristics are rarely employed for nurse rostering problems as they depend on the coverage constraint being a soft constraint requirement. In a very similar manner to the categorization of swap heuristics, the first three sub-categories are generally independent of each other (e1-e3). The two sub-categories (e4 and e5) are sub-categories that change how an edit heuristic is applied. Both sub-category e4 and e5 can be identified for some edit heuristics. The only identified exception is the Cowling heuristic H7 which can be categorized as a e3, e4 and s3 LPH under this categorization as, e3(e4)+s3.

The e1 sub-category heuristics if successively applied are able to fulfill the role of a e4 sub-category specialization. This is because it is possible for the addition of shift assignments and the removal to satisfy the coverage constraint requirement. The issue is adding and removing shifts can increase the solution search space, if one of the two LPHs in this category is applied too often.

The e2 sub-category deals with only changing a single shift assignment's shift type. It is a bit easier to satisfy a coverage constraint using this operator as it would only result in excess in the number of shift types assigned.

The e3 sub-category has LPHs that search a large search space. Changing a large number of shift types would have a larger effect compared to the sub-category of e2.

Three of the edit heuristics that fall into the sub-category e4 were used in an attempt to satisfy the coverage constraint. The two other edit heuristics that fall into this sub-category were used to change shift types depending on the skill type required for working the shift. This sub-category for edit heuristics allows the edit heuristics to focus on a smaller area of the search space in the same way it does for the local heuristic sub-category s4.

The Cowling heuristics fall mainly into the sub-category of e5. It could be that by having an additional requirement of improvement to soft or hard constraint violations, further focuses the search. If the search space does prove large then additional move acceptance may be necessary to focus the search. Additional move acceptance may be best used with LPHs which perform a lot of change to the solution space.

Table 6.3 Hyper-heuristics and edit heuristic categorization of LPHs used

Hyper-heuristics	Edit heuristic sub-categories
Bilgin et al. [210]	e1, e2, e2(e4)
Cowling et al. [205]	e3, e3(e4), e3(e5), e3(e4,e5), e3(e4)+s3
Burke et al. [208]	e3, e3(e4), e3(e5), e3(e4,e5), e3(e4)+s3
Bai et al. [209]	e3, e3(e4), e3(e5), e3(e4,e5), e3(e4)+s3

Table 6.3. Gives a summary of the edit heuristic sub-categories identified in selection perturbative hyper-heuristics studies.

6.3 Critical analysis of Nurse Rostering and Hyper-heuristics

The first section provides a critical analysis of the nurse rostering problem and the use of selection perturbative hyper-heuristics for solving the problem. The second section critically analyses the LPHs used by selection perturbative hyper-heuristics for solving the nurse rostering problem.

6.3.1 Nurse rostering problem

Nurse rostering is a challenging problem and has a number of areas that are yet to be explored. This is because the nurse rostering problem differs largely between hospitals, even within the same country. An example of this challenge is in some problem instances the coverage constraint is a hard constraint and in others it is a soft constraint. In problem instances where the coverage constraint is a soft constraint allows the selection perturbative hyper-heuristics applied to these problem instances to use LPHs that change shift assignments without creating an infeasible schedule. This makes it difficult to ascertain which LPHs are useful for different nurse rostering problems. Generally the most used LPHs were those that perform swaps within an existing roster; the approaches using these heuristics were also among the most successful. In this chapter these heuristics were categorized into the swap heuristic category.

The nurse rostering problem in some cases has been used as a problem domain to support the claim of generality gained by using a hyper-heuristic. The CHeSC2011 competition used the nurse rostering problem as one of the problem domains for competitive testing of the generality of selection perturbative hyper-heuristics across domains. While CHeSC2011 brought some new attention to nurse rostering this was only because it was one of the problem domains used for testing new selection perturbative hyper-heuristics. CHeSC2011 brought new attention to hyper-heuristics in general as well. The hyper-heuristic by Misir et al. [216] performed the best on average in the CHeSC2011 competition but performed poorly for the nurse rostering problem. The variable neighbourhood search hyper-heuristic by Ping-Che et al. [217] was found

to be the best performing hyper-heuristic overall for the nurse rostering problem but it was the worst performing hyper-heuristic for the bin-packing problem. These findings support the 'no free-lunch' theorem [221].

Selection perturbative hyper-heuristics for the nurse rostering problem have been shown to outperform meta-heuristics generally despite the higher level of generality achieved by a selection perturbative hyper-heuristic. Selection perturbative hyper-heuristics are never as bespoke as meta-heuristic approaches.

There has been little work in applying multi-point searches for solving the nurse rostering problem, especially evolutionary algorithms to solving the nurse rostering problem but even less work in performing a multi-point search of the heuristic search space. The only existing literature for searching a population of LPHs was early work using harmony search as a hyper-heuristic. In the published study only a handful of problem instances have been tested and the performance did not seem to be competitive with the state of the art. It was also found that many of the methods used by the selection perturbative hyper-heuristics submitted for the CHeSC2011 competition contained aspects of multi-point search methodologies such as keeping multiple candidate solutions and or improving upon them. For the reasons above, it is worth investigating an evolutionary algorithm as a selection perturbative hyper-heuristic.

6.3.2 Low-level heuristics

The choice and development of LPHs for the nurse rostering problem is a challenging area. There is no standard set of LPHs, as can be seen from literature in this chapter in section 6.1. Furthermore the LPHs have been used in different selection perturbative hyper-heuristics are applied to different problem instances. It is hard to determine if the selection perturbative hyper-heuristic is effective at using the LPHs or whether the set of chosen LPHs is effective for the problem instance(s). There has been consistency in the studies by Cowling et al. [205], Burke et al. [208] and Bai et al. [209] where the same set of LPHs was used in each. The Cowling LPHs were used for a nurse rostering problem where the coverage constraint was a soft constraint. This makes it difficult to use those LPHs in other problem instances where the coverage constraint is a hard constraint. The LPHs used by the Hyflex framework are quite complex but the same set is available across selection perturbative hyper-heuristics implemented using the framework. In Hyflex the issue is that there is no requirement to use all provided LPHs and also the LPHs require additional parameter values.

There are only a handful of studies investigating selection perturbative hyper-heuristics for the nurse rostering problem. Selection perturbative hyper-heuristic that have been used for the nurse rostering problem have been found to outperform meta-heuristic approaches [109], [209], [210]. Furthermore selection perturbative hyper-heuristics have been found to be effective for other combinatorial optimization domains also outperforming meta-heuristics for a variety of problem domains. In addition to this CHeSC2011 showed that innovative selection perturbative hyper-heuristics showed promise for solving the nurse rostering problem. This warrants further research into different types of selection perturbative hyper-heuristics for nurse rostering.

6.4 Summary

This chapter presents hyper-heuristic literature where the nurse rostering problem was solved. It also gives a categorization of the low-level perturbation heuristics used in literature and finally a critical analysis of nurse rostering and selection perturbative hyper-heuristics.

Chapter 7 Methodology

This chapter sets out the methodology used to achieve the objectives of this dissertation described in Chapter 1. Section 7.1 presents a critical analysis of related literature. Section 7.2 presents the research methodology to be followed. Section 7.3 details how each objective will be achieved. Section 7.4 gives a brief overview of the nurse rostering problem being solved. Section 7.5 presents the problem instances being used and the parameter settings. Section 7.6 describes the hypothesis testing which will be used in the study. Section 7.7 details the hardware and software used to achieve the objectives. Finally section 7.8 gives a summary of the chapter.

7.1 Critical analysis of related literature

From the survey of literature done in Chapter 4 and Chapter 5, it can be seen that there has been little research into solving the nurse rostering problem using selection perturbative hyper-heuristics. It can also be seen that the majority of the work involves single-point search methodologies. Multi-point search hyper-heuristics have been found to be effective in problem domains such as examination and school timetabling [30]. Examples of other multi-point searches have been effective at solving problems like course timetabling and project presentation scheduling [31], [34]. These have generally employed some form of evolutionary algorithm, usually a genetic algorithm with an indirect representation, this can also be referred to as a selection perturbative hyper-heuristic. The results of existing selection perturbative hyper-heuristics demonstrate that searching the heuristic search space is more effective than directly searching the solution space. The nurse rostering problem has a very large solution space. This suggests an evolutionary algorithm selection perturbative hyper-heuristic may be effective.

The major contribution of this dissertation is to investigate the generative perturbative hyper-heuristic for the nurse rostering problem. There has been little work with this type of hyper-heuristic generally and none for the nurse rostering problem. Generative perturbative hyper-heuristics have generally not been applied to timetabling and scheduling problem domains. The studies surveyed for solving the nurse rostering problem defined in this dissertation, used mostly meta-heuristics using neighbourhood operators and selection perturbative hyper-heuristics for finding solutions to the problem. This suggests there may even be a need for the generation of new perturbation heuristics for the nurse rostering problem.

The objectives as stated in Chapter 1 for the dissertation are: To investigate a genetic algorithm selection perturbative hyper-heuristic for solving the nurse rostering problem, to develop and analyse a genetic programming generative perturbative hyper-heuristic for generating LPHs to solve the nurse rostering problem and to compare the two developed hyper-heuristics' performance for the nurse rostering problem.

7.1.1 SPHH Justification

It is necessary to investigate selection perturbative hyper-heuristics first. In the literature (see Chapter 6 section 6.1) for solving the nurse rostering problem, all hyper-heuristics studied have been selection perturbative hyper-heuristics. These have been effective at solving the nurse rostering problem. There has been limited research for multi-point hyper-heuristics for solving the nurse rostering problem. This will be one of the goals of this research.

For the investigation of a selection perturbative hyper-heuristic a genetic algorithm will be used. The genetic algorithm selection perturbative hyper-heuristic will use a string representation where each character in a string representation represents a LPH that will be applied to a candidate solution. This is based on representations used by Han et al. [31] and Raghavjee [30]. An adaptive length representation will be used as Han et al. [31] found an improvement over previous work using a fixed length chromosome. A generational control model will be used as it will be a good way to establish a basic performance of the approach. It is well known control model and does provide a level of reliability. The initial population will be generated between a minimum and maximum size which will be based on the number of nurse shifts required for the problem instance. The maximum size of the chromosome will be decided based on memory limitations of systems used for simulations. This will ensure that there is little to no restriction on the size of chromosomes during the search. Tournament selection will be used as it is more computationally efficient compared to roulette wheel selection [13]. The crossover operator used will be the cut and splice crossover because it has been found to be effective for other selection perturbative hyper-heuristics in different problem domains. The mutation operator will be decided during development as there are different types of mutation to consider.

The LPHs used will be from the swap heuristic category as defined in section 6.2.1. This is primarily because the problem instances use the coverage constraint as a hard constraint requirement. The LPHs used will be from the s1, s2 and s3 sub-categories with some falling into the s4 and s5 categories if necessary. Previous work applied to the same problem instances has successfully used LPHs from the s3 sub-category. From the categorization of LPHs for nurse rostering in section 6.2.1 has shown that the s1 and s2 sub-categories should be utilized for selection perturbative hyper-heuristics in this problem domain. This is because these are the most commonly used swap heuristics. The s4 and s5 sub-categories will be considered for use during the study as it is not clear whether these will be advantageous.

7.1.2 GPHH Justification

The investigation for the generative perturbative hyper-heuristic will use genetic programming and strong typing. There has been no previous work in generating LPHs for the nurse rostering problem. However the surveyed literature for solving the nurse rostering problem has shown that perturbation heuristics are effective for solving the nurse rostering problem. It follows that evolving new LPHs should be attempted.

Genetic programming has been shown to be good at the generation of low-level heuristics as a generative hyper-heuristic [221]. Strong typing will enforce the relationships between primitives; this is similar to using a grammar. It should be stated that grammatical evolution could perform a similar function but it does search the program space in a different way [222]. The study will be influenced by the study by Sabar et al. [204] and the study by Nguyen et al. [196]. The generational control model will be used; it is a commonly used control model for genetic programming. The grow method will be used to produce the initial population. The initial population generation method generally does not seem to affect fitness [223]. The grow method provides diverse trees and will be sufficient for the purposes of investigating a generative perturbative hyper-heuristic for the nurse rostering problem. Tournament selection will be used as it is the most computationally efficient method. The generative perturbative hyper-heuristics will be investigated as it could find new perturbation heuristics for the nurse rostering problem, and or facilitate identification of effective components for perturbation heuristics for the nurse rostering problem. Standard crossover and sub-tree mutation will be used because these offer a good variety of exploration and exploitation. Crossover provides a local search of the program space. Sub-tree mutation allows for a global search of the program space. Crossover and Sub-tree mutation have been found effective for genetic programming. Other genetic operators will be considered during the study.

The function set will contain move acceptance methods such as, simulated annealing and great deluge for move acceptance, simple move acceptance criteria, functions to combine terminals and functions. These will be used based on what has been done in other generative perturbative hyper-heuristics. Bilgin et al. [213] explored move acceptance for a selection perturbative hyper-heuristic for nurse rostering and found generally great deluge and simulated annealing move acceptance (meta-heuristics) were equivalent to accepting any equal or improving move. Across other problem domains meta-heuristic move acceptance methods perform better although they have a higher computational cost. For this reason and to provide options to the genetic programming approach a number of move acceptance methods will be made available. The move acceptance methods include: Accept improving moves only, accept improving or equal moves, great deluge, simulated annealing, adaptive iterative-limited list acceptance (AILLA) and late acceptance hill climbing. Unlike the study by Sabar et al. [204], IF conditionals will be used similar to those used by Nguyen et al. [196] and an iteration function will be made available. The terminal set will include LPHs of sub-categories s1, s2, s3, s3(s4), s1(s4) and s2(s4). The sub-category s5 is redundant because of the inclusion of the move acceptance function(s).

7.2 Research methodology

The objectives will be achieved using the proof by demonstration methodology [224]. This research methodology for computer science requires the development of a single approach which is iterated upon to achieve the stated objective(s). Firstly an initial approach must be developed. The initial approach is based on an analysis of the literature. The initial approach will then be improved upon using iterative refinement. Changes to the approach will be based on testing. Iterative improvement occurs until an improvement is made or an improvement cannot be feasibly made. For each iteration the revision of the approach is evaluated in terms of the objectives of the problem to be solved. These approaches should be produced to meet the objectives of the research question.

In order to analyse the developed approaches data will be collected. This will be done by using a problem domain benchmark data set as a measure. It will also be compared empirically to other method's available results from the same problem domain benchmark data set. Finally the empirical data collected and observations made by the researcher will be presented in a conclusion.

7.3 Objectives

This section describes how the research methodology will be implemented in order to achieve the three objectives of this dissertation. Objective one and objective two are similar and approached in the same manner; this methodology will be discussed in section 7.3.1. Objective three requires completion of both objectives one and two which will be discussed in section 7.3.2.

7.3.1 Objective one and two

The first objective set in Chapter 1, is to investigate a genetic algorithm as a selection perturbative hyper-heuristic for the nurse rostering competition. The second objective set in Chapter 1, is to investigate the creation and performance of a generative perturbation hyper-heuristic model for the nurse rostering problem. These objectives are achieved through the proof by demonstration methodology.

Firstly an initial approach will be developed for each objective, descriptions of the initial approaches can be found in section 7.1.1 for objective one and section 7.1.2 for objective two.

The two developed approaches will be tested using the nurse rostering competition benchmark data set described in section 7.4.1.

The developed approaches will be refined until no improvement can be made or an improvement is not going to provide significant improvement for the work required or there is no significant improvement possible.

The refinements made to both approaches will be similar as they are both evolutionary algorithms as described in Chapter 2. Refinements will include looking at the initial population generation, the genetic operators, the selection of parents and the control model.

Objective one requires analysis of the genetic algorithm and the performance of the genetic algorithm. Objective two requires the analysis of how the LPHs were created, how the LPHs performed and the structure of these LPHs.

SPHH will be executed 30 times for each instance in the INRC2010 benchmark data set. GPHH will be executed 30 times per LPH to evolve. A number of LPHs will be evolved. Each evolved LPH will then be executed on the INRC2010 benchmark set using a time limit as a stopping criterion.

7.3.2 Objective three

The achievement of objective three requires the completion of both objective one and objective two. Both approaches will be compared using statistical hypothesis testing as shown in section 7.6.

7.3.3 Measurements for analysis of the objectives

There is currently no standard measurement to compare the results of hyper-heuristics. Most literature compares results in terms of feasibility and quality. Feasibility is measured by hard constraint violations, a feasible solution has no hard constraint violations. If hard constraint violations do occur they will be penalized heavily. Quality is measured using the minimum soft constraint violations (minSCV) obtained by the method.

Results will be compared using the number of hard and soft constraint violations. The measurements taken will be: hard constraint violations, minSCV, the average soft constraint violations over 30 runs (avgSCV), the standard deviation of the average soft constraint violations and the number of generations taken.

For objective 1, the number of generations taken will be discussed; this is how many times the population was changed while a combination or combinations of LPHs were applied to candidate solutions. The minSCV will be compared to the best known result (BKR) for each instance.

For objective 2, the number of generations taken will be discussed; this is how many times the population was changed until an LPH is evolved. The minSCV will be compared to the BKR for the evolved heuristics. The avgSCV will also be used to measure the performance of the evolved heuristics.

For objective 3, the minSCV of both methods will be compared. The avgSCV and standard deviation will be used for hypothesis testing of the results of SPHH to GPHH.

7.4 The nurse rostering problem

The first international nurse rostering competition benchmark data set [96] will be used to test SPHH and GPHH. This benchmark data set contains 60 instances each with different work contracts for the nurses. Instances are subdivided into tracks based on the number of nurses. These are sprint, medium and long. Each track had instances released under different periods e.g. early instances were released first, late instances were released near the end of the competition and hidden instances were unseen to the competitors. For competition purposes the sprint track was allowed 8 seconds, the medium track 8 minutes and the long track 10 hours. The time limits were obtained using a provided CPU benchmarking tool. For SPHH these limits were ignored as it is known that evolutionary algorithms and hyper-heuristics generally require more runtime than single-point search approaches. For GPHH these runtimes will be used as a guideline for the final evaluation of evolved LPHs (Chapter 4 section 4.3). The INRC2010 benchmark instance details can be found in Chapter 4 section 4.3 in Table 4.1.

7.4.1 Justification for benchmark set

While NSPLib offers an impressive amount of data that can be used to perform statistical tests, it does not feature complex and modern constraints such as unwanted shift patterns. The main reason for not using this benchmark set is the lack of comparative studies in the field of hyper-heuristics. At least one author has expressed difficulty in accurately comparing their work with the available results [225].

The INRC2010 benchmarks provide comparison of competition proven results (Under rules and a time limit) as well as results submitted post-competition (no time limit verification). This provides a comparison against a variety of different algorithms that are new or state of the art. INRC2010 does feature a constraint that is not present in the Nottingham benchmarks which is the unwanted patterns constraint which is a slightly more specific version of the shift succession constraint as you can specify unwanted patterns for certain days. However the Nottingham instances have a wide number of constraints to solve since they include time restriction constraints (where hours between shifts matter), these (for example) are not covered in the INRC2010 benchmark set. INRC2010 is considered to have more modern and standardized constraints.

7.5 Problem instances

GPHH requires the evolution of LPHs. These LPHs will be evolved using a seen instance, the seen instance is the only instance used during the evolution of an LPH. Once evolved, each LPH is applied to all instances (seen and unseen). In order to ascertain the effects of using different seen benchmark instances for the evolution of LPHs, 9 INRC2010 instances will be used to evolve 9 LPHs. This will be a random selection of an early, late and hidden instance from each track (3 sprint instances, 3 medium instances and 3 long instances). This will be done to use instances which have different constraints. In order to discover if evolving a LPH using more than one seen instance is advantageous, a further four LPHs will be evolved using sets of three seen instances. The first three will select an instance randomly from each track for early, hidden and late instances. The last will take three instances randomly from the sprint track.

Each evolved LPH is assigned a label based upon the instance used to evolve the LPH. For example the LPH evolved using medium_early05 is given the label **ME**. The instances and associated labels assigned to the 13 heuristics which will be evolved using these instances, can be seen in Table 7.1.

Table 7.1 Labels assigned to evolved LPHs using corresponding instances

Evolved heuristic	Seen instance(s)		
SE	sprint_early5		
SH	sprint_hidden4		
SL	sprint_late6		
ME	medium_early5		
MH	medium_hidden2		
ML	medium_late3		
LE	long_early4		
LH	long_hidden5		
LL	long_late2		
S	sprint_early2	sprint_hidden1	sprint_late2
L	sprint_late10	medium_late4	long_late3
H	sprint_hidden6	medium_hidden5	long_hidden3
E	sprint_early9	medium_early4	long_early4

7.6 Hypothesis testing

A two-tailed hypothesis test will be used to determine the statistical significance of results obtained when comparing the performance of SPHH and GPHH. This test can only be used with results that can form a normal distribution that is there must be 30 or more in the two samples tested. Z-tests are used to calculate a Z-value, this Z-value is compared to the critical value to determine the level of significance. The level of significance and corresponding critical value and decision rules are given in Table 7.2. In this dissertation the z-test will be applied with a value of $\alpha = 0.05$, or 95% significance. Testing for a statistically significant result requires the formulation of a null hypothesis and an alternate hypothesis:

$$H_0 : \mu A = \mu B$$
$$H_a : \mu A > \mu B$$

If the calculated value of Z is less than the critical value there is no statistical difference in comparing the means and the null hypothesis (H_0) is accepted. If the Z value is greater than the critical value than the alternate hypothesis is accepted (H_a) and there is a statistically significant difference in the two means. A negative Z value represents that method **A** is worse than method **B**.

Table 7.2 Levels of significance, critical value and decision rules for Z hypothesis test

Significance (α)	Critical value	Decision rule
0.01	2.33	Accept H0 – IF Z < 2.33
0.05	1.64	Accept H0 – IF Z < 1.64
0.1	1.28	Accept H0 – IF Z < 1.28

7.7 Technical specifications

The two proposed hyper-heuristic approaches, a genetic algorithm selection perturbative hyper-heuristic and a genetic programming generative perturbation hyper-heuristic will be developed using the Java programming language. The Netbeans integrated development environment will be used. The linear congruential random number generator native to the Java programming

language was used to generate random numbers. The approaches will be developed on a computer with the following specification: Intel i7 2600 3.3GHz, 4 GB of RAM running the Windows 7 operating system.

Simulations were run using:

- Intel i7 2600 3.3GHz, 4 GB of RAM running Linux using the distribution Fedora 17
- Hewlett Packard z820, Intel Xeon E5-2697v2 2.7GHz, 512GB of RAM running the Windows 7 operating system
- Center for high performance computing sun cluster, see: www.chpc.ac.za for further details

7.8 Summary

This chapter presents the methodology used for achieving the objectives outlined in **Chapter 1**. The measurements used to analyse the achievement of the objectives, an overview of the nurse rostering problem and justification for the use of the chosen benchmark set is given. SPHH and GPHH will be applied to the INRC2010 benchmark data set. The problem instances required for the creation of LPHs by GPHH are presented. In cases where there is need to show the significance of a result a two tailed hypothesis test will be used to show statistical significance. Finally the technical specifications for the development and testing of the developed approaches (SPHH and GPHH) are provided.

Chapter 8 Genetic Algorithm Selection Perturbative Hyper-Heuristic

This chapter presents SPHH, a genetic algorithm using an indirect representation as a selection perturbative hyper-heuristic. This is a multi-point hyper-heuristic approach. This genetic algorithm implemented was influenced by genetic programming and genetic algorithm hyper-heuristics. This genetic algorithm hyper-heuristic uses a generational control model [20]. Genetic operators are used to search a space of LPHs. The chapter details the SPHH algorithm, the representation used and LPHs used by the genetic algorithm, the selection methods used, the genetic operators, the multithreading approach used and the parameters chosen.

8.1 SPHH Algorithm

The genetic algorithm used for SPHH is presented in Algorithm 8.1. Firstly an initial population is created, this population is then refined iteratively following a process of evaluation, selection of parents and use of genetic operators to create offspring which form the population for each generation. Section 8.2 describes the initial population creation, representation and LPHs used, Section 8.3 presents the selection and evaluation of individuals for genetic operators, section 8.4 describes the genetic operators used, section 8.5 discusses using multithreading to optimize the SPHH algorithm and section 8.6 gives the final parameters used for the algorithm.

1. Create initial population
2. Evaluate individuals in population
3. Create new generation by:
    a. Apply each heuristic in the individual to the current solution
    b. Set best individual's solution as the current roster
    c. Select parents using tournament selection
    d. Apply genetic operators to selected parents and evaluate individuals in population
4. Repeat 3. until a maximum number of generations has been reached or the solution has converged. Return the best candidate solution

Algorithm 8.1 Genetic algorithm hyper-heuristic

8.2 Representation and initial population generation

An individual in the population is represented by a string, this string is made up of characters. Each character represents a LPH. A number of heuristics were developed based on literature. Heuristics that swap multiple days must perform at least two swaps. The LPHs are listed below with the character used to represent the LPH, its categorization (refer to section 6.2) and with a description of the LPH. For example a heuristic which swaps a shift type between nurses will be in the category of s1.The LPHs used included the following perturbation heuristics:

- T (l1(l5)) –Randomly select two nurses and a day where both have worked shifts, swap the shift types between the two nurses. Attempt for *n* times and return the best found solution.

- Z (l1(l5)) –Randomly select two nurses and a day where both have worked shifts, swap the shift types between the two nurses. Attempt for *n* times but stop when an improved solution is found.
- A (l2(l5)) –Select a day randomly, randomly select a nurse working a shift and another nurse working a free shift for the selected day. Swap the shift type to the nurse with the free shift for the selected day. Attempt for *n* times and return the best found solution.
- Q (l2(l5)) –Select a day randomly, randomly select a nurse working a shift and another nurse working a free shift for the selected day. Swap the shift type to the nurse with the free shift for the selected day. Attempt for *n* times but stop when an improved solution is found.
- Y (l3(l5)) –Select two nurses randomly, select a random subset of days in the scheduling period. Swap the associated shift types for the selected day. Attempt for *n* times and return the best found solution.
- X (l3(l5)) –Select two nurses randomly, select a random subset of days in the scheduling period. Swap the associated shift types for the selected days. Attempt for *n* times but stop when an improved solution is found.
- W(l3(l5)) – Select two nurses randomly, select a subset of weekdays randomly. Swap the associated shift types for the selected days. Attempt for *n* times and return the best found solution.
- F (l3(l5)) – Select two nurses randomly, select a subset of weekdays randomly. Swap the associated shift types for the selected days. Attempt for *n* times but stop when an improved solution is found.
- E (l3(l5)) – Select two nurses randomly, select a subset of weekend days randomly. Swap the associated shift types for the selected days. Attempt for *n* times and return the best found solution.
- C (l3(l5)) – Select two nurses randomly, select a subset of weekend days randomly. Swap the associated shift types for the selected days. Attempt for *n* times but stop when an improved solution is found.
- M (l2(l4)) – Randomly select a nurse which has a requested shift off (see section 4.1) violation, randomly select a nurse which has a free shift for the same violation. Assign the nurse with the free shift the shift type of the first nurse and remove the first nurse's shift assignment.
- K (l2(l4)) – Randomly select a nurse which has a requested day off (see section 4.1) violation, select a nurse which has a free shift for the same violation. Assign the nurse with the free shift the shift type of the first nurse and remove the first nurse's shift assignment.
- B ((l1+l2)(l5)) – Randomly select a nurse, a list of nurses is created randomly, a day is randomly selected. The first nurse's associated shift type is swapped with each nurse in the list. Improving moves are kept.
- b – Blank move, this means no heuristic is executed at this point in the string. Anwar et al. [214] also provided a LPH that does nothing.

Each character of the string is randomly chosen from the given set of LPHs. An example of an individual is: "*TZTMKB*" the first LPH to apply would be 'T' and the last would be 'B'. The candidate solution obtained from applying these LPHs to the current schedule would be compared with the best schedule obtained by SPHH and if it has lower soft constraint violations it would replace the best schedule obtained. This is done for each individual in the population. Hard constraint violations are avoided as the LPHs do not violate the single shift per day hard constraint and cannot add excess shifts so they cannot violate the coverage requirements.

In the creation of the initial population, the length of each string is randomly generated within a lower value and an initial upper value. The number of shift assignments detailed in the cover requirements for the scheduling period (see Chapter 4 section 4.1) is multiplied by a number for

the lower and a number for the upper bound to determine these bounds. For example an initial population is generated and the lower value is set to 5, the initial upper value to 10 and the number of shift assignments is 200. Then the initial population would only contain individuals uniformly with string lengths of 1000 and 2000. The values used for upper and lower values were chosen during the study. From the first generation, the upper value is set to a maximum limit, this limit is high but safely avoids any system memory limitations. A high initial upper value during the initial population generation affects performance, as higher string lengths directly increase time it takes to evaluate individuals.

8.3 Evaluation and selection

Before the SPHH is used as a solver for the NRP, an initial schedule is created by randomly allocating shift types to nurses. The number of shift types allocated is the number of shifts specified in the coverage requirements for the scheduling period. Only a single shift is allowed to be allocated to a nurse, for each working day. This avoids incurring hard constraint violations for the chosen problem instances. Each individual in the population is applied to the same initial schedule. The soft constraint violations are the measure of fitness of an individual. For each generation the best improvement on the initial schedule found after all individual's execution replaces the initial schedule. This is a form of shared memory as information is given to all the individuals in the population.

1. Select a random sample using the parameter tournament size
2. Set the best individual (*i*) as the first individual in the sample
3. Repeat until there are no individuals to compare against:
    a. Compare the fitness of *i* with *i*+1 from the sample
    a. **If** the fitness of *i* is higher than *i*+1, replace *i* with *i*+1
4. Return *i*

Algorithm 8.2 Tournament selection

Tournament selection is used to choose the parents to create the next generation. This selection method returns the fittest individual of a random sample of individuals.
Algorithm 8.2 presents the process in detail. A random sample is simply individuals in the population chosen at random and put into a group. The size of this group is decided by the tournament size parameter.

8.4 Genetic operators

In each generation the population is created by applying mutation and crossover operators to the selected parents. Genetic operators are chosen based on application rates e.g. for 0.7 crossover and 0.3 mutation with a population size of 100, will create 70 offspring through crossover and 30 offspring through mutation in the next generation.

The crossover operator combines two individuals that are selected using tournament selection. Two random points, one in each individual is selected. These points are used to create substrings where the characters left of the point in the second individual are appended to the

characters right of the point in the first individual and the characters left of the point in the first individual are appended to the characters right of the point in the second individual. These two offspring are evaluated and the fittest becomes a member of the next generation. The process is depicted in Figure 8.1. The crossover operator is unable to produce offspring that are greater than the maximum string bound set due to memory limitations. If offspring are produced that exceed this bound, the characters after the limit are removed. The removal of characters occurs before the offspring are evaluated.

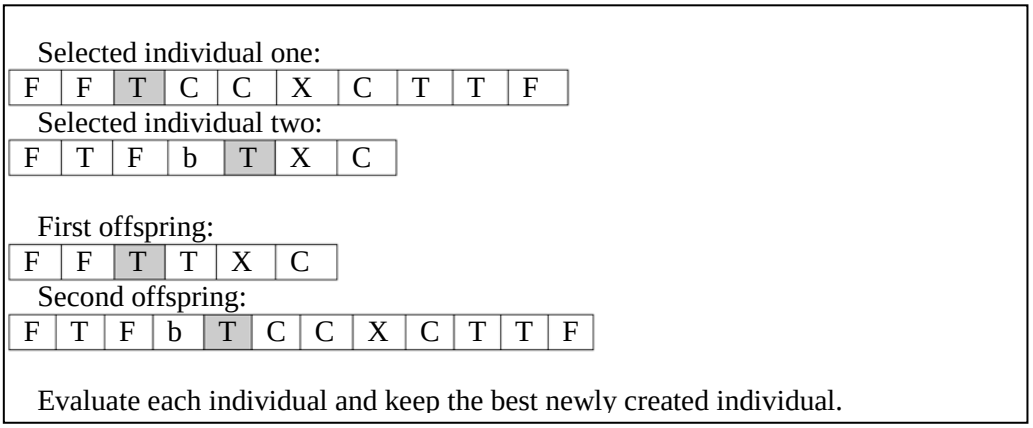

Figure 8.1 Crossover example

The mutation operator changes one individual selected using tournament selection. One random character in the string of LPHs is selected and changed to a randomly selected character. The new offspring is then evaluated. The process is depicted in Figure 8.2.

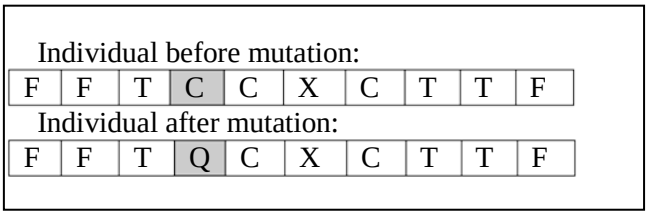

Figure 8.2 Mutation example

8.5 Multithreading

Multithreading was introduced to SPHH in order to improve runtimes. It also adds a bit of entropy in that each thread will execute in a non-deterministic order which means that a usually predictable pseudo random number generator will not produce the same sequence of random numbers with the same seed.

Initially the evaluation process was threaded where constraint evaluations were put on threads and then added to the thread pool however this did not result in performance gains as the work load of most of the constraint evaluations was minimal.

Evolutionary algorithms are an embarrassingly parallel problem. Where the individuals of the population are processed by creating offspring, this process is only dependent on choosing parents from tournament selection. There is no reason why the genetic operators themselves cannot be called in parallel. This allows most of the algorithms runtime to be reduced significantly based on Amdahl's law in Equation 8.1 which states that a speed up (S) is related to the number of threads(n) and the amount of processing that is required to be linear (B). Since

the majority of the work can be performed in parallel, B can be assumed to be less than 10% of the algorithm. The machines used for testing have access to 8 threads. Giving an estimated potential reduction in runtimes by as much as 92%. Each call to a genetic operator is executed on a separate thread.

$$S(n) = \frac{1}{B + (1 - B) - n}$$

Equation 8.1 Amdahl's law

8.6 Parameters

Table 8.1 details the parameters used for SPHH simulations. The population size was decided during the study by testing different values. The initial length of the individuals in the population is randomly set between an initial maximum value and a minimum value. These values were scaled to the coverage requirements for each instance. After the initial population generation the maximum individual length is set to a large integer value. The tournament selection size was set to 5 individuals in the population size promoting selection of better performing individuals. The genetic operator rates were decided during the study, through observation of different combinations. Shared memory was used because during the study it showed improved results. The convergence limit is the number of generations for which the fitness of the candidate solution is unchanged, after this limit is reached SPHH terminates. The convergence limit was decided during the study based on an observation that generally convergence occurs within 25 generations. A 100 generation limit was set as a generous limit. The low-level heuristic set (see: Section 8.2) was influenced by related literature as a result of the discussion in Chapter 6 Section 6.3.2.

Table 8.1 Parameters used for SPHH runs

Parameter	Value
Population Size	100
Initial maximum individual length	25 × Coverage requirements
Minimum individual length	10 × Coverage requirements
Maximum individual length	1342177
Tournament size	5
Crossover rate	0.7
Mutation rate	0.3
Shared memory	Yes
Benchmark instance data set	The First Nurse Rostering Competition Instances
Convergence limit	25
Low-level perturbation heuristic set	T,Z,X,Y,F,W,C,E,Q,A,K,M, B,b
Generational limit	100

8.7 Summary

This chapter has presented SPHH a multi-point selection perturbative hyper-heuristic. The chapter describes the SPHH algorithm, the representation used and the creation of individuals in the population. The details of the LPHs used are provided. The chapter then covers the selection and evaluation methods, the genetic operators used and multithreading which was necessary to improve runtime performance. Finally the chapter presents the parameters used for the SPHH approach simulations.

Chapter 9 Genetic Programming Generative Perturbation Hyper-Heuristic

This chapter presents GPHH, a strongly typed genetic programming algorithm as a generative perturbative hyper-heuristic. This genetic programming algorithm was influenced by the state of the art of generative perturbative and generative constructive hyper-heuristics. The genetic programming algorithm studied here uses a steady-state control model. Genetic operators are used to combine function and terminal set elements into new low-level perturbative heuristics. The chapter describes the GPHH algorithm, the function and terminal set used, the initial population generation methods and representation used, genetic operators and the chosen parameters.

9.1 GPHH Algorithm

This section presents the GPHH algorithm. Strong typing is used to create more structured trees by limiting the arguments of certain operators. The genetic programming approach uses the steady-state control model. In previous research it was found that the steady-state control model was an improvement compared to the generational control model, as it prevents weak individuals from being entered into the population [25]. The steady-state control model was also found to reduce premature convergence. During the study, the generational control model was attempted but performed poorly.

Algorithm 9.1 provides an overview of the GPHH. First an initial population is created and is evaluated. Then at each iteration a set number of offspring are created using a selected genetic operator based on a set probability. The parent(s) required for the genetic operator are selected using tournament selection. The newly created offspring are evaluated. Once the set number of offspring have been created they replace individuals in the population provided they do not exist as duplicates are not allowed in the population. Inverse tournament selection is used to replace individuals in the population. This algorithm repeats until a set number of iterations are reached.

1. Create initial population
2. Evaluate individuals in population (See Algorithm 9.2.)
3. Create a number of individuals by:
    a. Probabilistically select genetic operator type
    b. Select parents using tournament selection
    c. Apply genetic operators to selected parents and evaluate newly created individuals
    d. Replace individuals if they are not present in the population using inverse tournament selection.
4. Repeat until a maximum number of generations.
5. Return the best program

Algorithm 9.1 Genetic programming algorithm overview

Algorithm 9.2 shows an overview of how each individual in the population is evaluated during the execution of Algorithm 9.1. Firstly the parse tree is interpreted as a heuristic. This heuristic is executed 10 times, this tests the individual on randomly generated initial solutions, this prevents over fitting. Each execution applies the heuristic to the candidate solution for the instance, until a set time limit is reached. Only one instance is used for the 10 executions. The individual is given a fitness value by using the soft constraint values obtained over these 10 executions. This is the average fitness multiplied by the minimum fitness. This was chosen as it rewards individuals that find minimum results quickly while also penalizing individuals with performance that varies by a large amount.

1. Interpret parse tree
2. Repeat for 10 executions
    a. Create random initial solution
    b. Execute parse tree for time period *n*
3. Update fitness of individual

Algorithm 9.2 Evaluation phase of GPHH individual

9.2 GPHH terminal and function set

GPHH aims to create new LPHs. These are composed of existing move acceptance methods and LPHs. LPHs make changes to an initial solution. Genetic programming is used to combine these parts which have been divided into a function set and terminal set. The created LPH is applied to randomly generated initial solutions. The default move acceptance for an individual is to accept all moves. As the parse tree is traversed, each move acceptance method will replace the current move acceptance method, when applying terminals to the initial solution. Some move acceptance methods have a number of parameters. These parameters are not reset until the initial solution is changed. Each LPH is applied multiple times to the solution for a limited period. This is done to test the effectiveness of the evolved LPH.

The terminal set consists of 14 LPHs. The LPHs chosen for the terminal set are based upon an analysis of the literature in Chapter 6 section 6.2. These were chosen to represent a wide range of possible LPHs to give the process of evolving a LPH more flexibility. This flexibility is achieved by being able to combine existing LPHs with move acceptance methods. These are presented in a bulleted list which has the letter '**n**' followed by a number for the LPH and the LPH categorization in brackets:

- n0 – No change to initial solution.
- n1(l1) – Randomly select two nurses and a day where both have worked shifts, swap the shift types between the two nurses.
- n2(l2) – Select a day randomly, randomly select a nurse working a shift and another nurse working a free shift for the selected day.
- n3(l3) – Select two nurses randomly, select a subset of days in the scheduling period randomly. Swap the associated shift types for the selected day.
- n4(l3(l4)) – Select two nurses randomly, select a random subset of weekend days. Swap the associated shift types for the selected days.
- n5(l3(l4)) – Select two nurses randomly, select a random subset of weekdays. Swap the associated shift types for the selected days.
- n6(l2(l4)) – Randomly select a nurse which has a requested shift off (see Chapter 4 section 4.1) violation, randomly select a nurse which has a free shift for the same violation. Assign the nurse with the free shift the shift type of the first nurse and remove

the first nurse's shift assignment.

- n7(l2(l4)) – Randomly select a nurse which has a requested day off (see Chapter 4 section 4.1) violation, select a nurse which has a free shift for the same violation. Assign the nurse with the free shift the shift type of the first nurse and remove the first nurse's shift assignment.
- n8(l1(l4)) – Select a nurse randomly, then select a nurse with a roster with a higher soft constraint cost. Randomly select a day where both have worked shifts, swap the shift types between the two nurses.
- n9(l1(l5)) –Selects two nurses randomly and a sequential period of days randomly. Shifts between these nurses are swapped if a set probability is randomly overcome.
- n10(l3) – Select two nurses randomly, select a sequential subset of days in the scheduling period randomly. Swap the associated shift types for the selected day.
- n11(l1(l4)) – Select two nurses randomly which have different contracts, and a day where both have worked shifts, swap the shift types between the two nurses.
- n12(l3(l4)) – Select two nurses randomly which have different contracts, select a subset of days in the scheduling period randomly. Swap the associated shift types for the selected day.
- n13(l3(l4)) – Select a nurse randomly, then select a nurse with a roster with a higher soft constraint cost. Select a subset of days in the scheduling period randomly. Swap the associated shift types for the selected day.

The function set contains four functions: Combiners (H and C), If statements, Move acceptance methods (A) and Iterations (I).

Combiners represent multiple statements in a program. There are two types of Combiners; high-level (H) and low-level (C). Each parse tree must begin with a high-level combiner. There are six high-level combiners with an arity of 1 to 6 (H1-H6). Arguments of high-level combiners can be any of the following functions: Repeat, move acceptance methods and low-level combiners. There are two low-level combiners; C2 which has an arity of 2 and C3 which has an arity of 3. These accept low-level combiners, If statements and terminals as arguments.

An example of the high-level combiner, **H3** can be seen in Figure 9.1. **H3** combines 3 branches. When the parse tree is executed, the branch beginning with **I** is run first, **I** is a repeat function so any functions and terminals making up the subtree will be executed ten times. Then the branch beginning with **A** will execute, **A** is a move acceptance method. The third branch, **C2** will execute the branches from left to right.

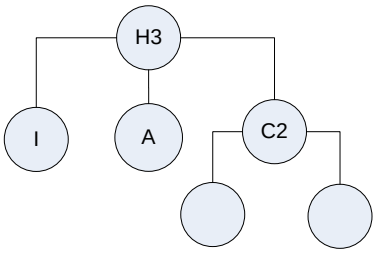

Figure 9.1 Example of high-level and low-level combiner functions

Iteration functions have an arity of 1. The arguments they accept are; move acceptance methods or low-level combiners. There are two iteration operators both repeat the branch for ten iterations. Different values were tried during development but ten worked well. The first, **I1** executes the branch and retains all changes made to the initial solution. **I1** is intended to

encapsulate an effective branch and repeat it. The second, **I2** stores each change to the candidate solution and applies the best change found at the end of ten iterations. **I2** is intended to find the best change to the candidate solution by a branch.

In Figure 9.2 it can be seen that there is an iteration function in the first branch of the high-level combiner **H3**. The iteration function has a move acceptance method **A** as an argument and is followed by **C2** which combines two LPHs as terminals. The LPHs are each applied using the same acceptance method **A**. The LPH, **n3** is applied to the initial solution and the move is accepted or rejected by the move acceptance method **A** and then n1 is applied to the candidate solution and the move is accepted or rejected by the move acceptance method **A**. This branch is repeated for ten iterations. Then the next branch containing a move acceptance method **A** is applied and finally the branch containing the low-level combiner **C2** is applied.

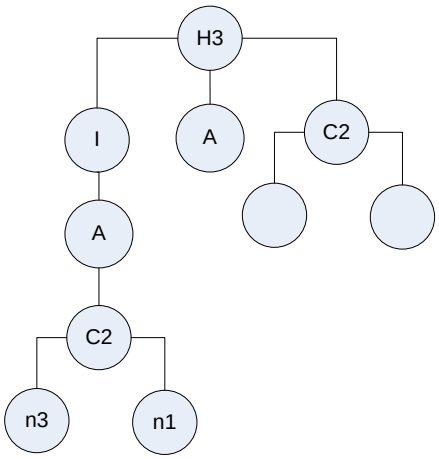

Figure 9.2 Example of an iteration function

Move acceptance methods have an arity of 1. The arguments they can take are low-level combiners, If statements and terminals. There are 8 move acceptance methods. The acceptance methods chosen are taken from literature. These are given in Table 9.1. Simple move acceptance methods such as; accept improving or equal moves were included. The intention of high-level combiners is to allow for multiple move acceptance methods to be present in the same parse tree. The first move acceptance method in the parse tree will be applied to all terminals (LPHs) until a branch with a different move acceptance method is reached, at this point that move acceptance method will be used.

The move acceptance methods; accept improving only, accept improving or equal, great deluge and simulated annealing for this problem were explored for a hyper-heuristic using random heuristic selection for the nurse rostering problem [213]. This study found improving only to be the worst acceptance method for their hyper-heuristic but found great deluge to be the best. However, no combination was found to be significantly better than another. Due to this even though ‘improving only’ seems to perform poorly it should be an option to the GPHH.

The methods of late acceptance [159], [226] and step counting [227] are algorithms proposed for improving the hill climbing acceptance method. The step counting algorithm was shown to perform generally better than simulated annealing and late acceptance when applied to the examination timetabling problem. These acceptance methods are state of the art and need to be options for GPHH.

The adaptive iterative limited list algorithm was introduced by Misir [228] it is similar to the late acceptance method but was shown to perform better than late acceptance, improving equal and great deluge when applied to the ready-mixed concrete delivery problem. This acceptance method was also used [213], [216] to create an adaptive hyper-heuristic for Hyflex [141]. This hyper-heuristic won the cross domain heuristic search challenge (CHeSC2011). Given the comparative performance to state of the art move acceptance methods it was included in the function set.

Table 9.1 Acceptance methods components

Acceptance Methods (type A)	Description
A1	All Moves
A2	Improving only
A3	Improving or Equal
A4	Late acceptance [229]
A5	Great deluge [70]
A6	Step counting [230]
A7	Simulated annealing [69]
A8	Adaptive Iterative Limited List Acceptance [216]

In Figure 9.3 the high-level combiner is **H2**. The first branch is an iteration operator followed by the move acceptance method **A3** (which accepts all improving and equal moves) followed by the low-level combiner **C2** which has branches **n3** and **n1**. These are executed sequentially using **A3** to accept or reject moves. When the ten iterations of the repeat operator are finished, the next branch **A5** changes the move acceptance method to the great deluge method. The next operator in the tree is **C2** which consists of **n9** and **n0**. The LPH **n9** is applied to the solution and the moves are accepted or rejected based on the great deluge method. **n0** is a null move but if it were a LPH that made a change to the solution it would be applied and the great deluge method would consider its changes to the solution.

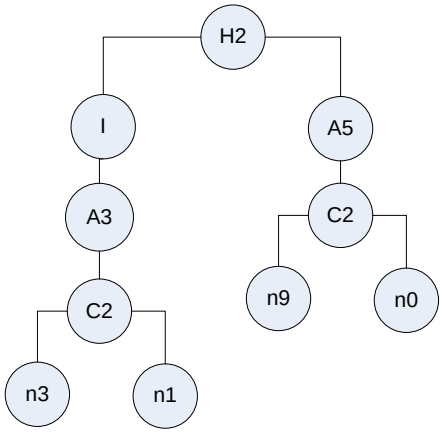

Figure 9.3 Move acceptance function example

There are two If statement operators, the first **IF-C** has an arity of 2. **IF-C** checks if the candidate solution has not changed during Algorithm 9.2. If it has not changed the alternate branch is executed otherwise the first branch is executed. This is intended to allow a LPH to evolve a branch to escape local optima. The second is **IF-I** which has an arity of 3. The first branch of **IF-I** is always executed, if an improvement in soft constraint violations are found then the second branch is executed otherwise the third branch is executed. This is intended so that an alternate branch can be executed if the result of applying the first branch does not effect change

on candidate solution. The arguments taken are: Low-level combiners, If statements and terminals.

In Figure 9.4 the high-level combiner is **H2** the first branch has an acceptance method of **A3** followed by the operator **IF-I**. **IF-I** applies the first argument to the candidate solution, once the result of applying the first argument is known, the conditional operator checks if this result is an improvement. If the result is an improvement the second branch is traversed, otherwise the third branch is traversed. In the example, the first branch of **IF-I** is applied to the candidate solution, in this example this would be the application of **n3**. If the move made by **n3** and accepted by **A3** is an improvement to the soft constraint violations of the solution then the second branch is applied (**n1**), otherwise the branch combining **n6** would be applied to the solution. The second branch of **H2** changes the move acceptance method to **A5** (great deluge) the following branch is **IF-C**. **IF-C** checks if the number of soft constraint violations has stayed the same for a number of iterations. If this is false **n9** is executed, if this is true the second branch combining **n0** is executed. In this example no change would occur however this would not always be the case with **IF-C** as it could combine any combination of **IF-I**, **IF-C**, **C2, C3** or any terminal (LPH)**.**

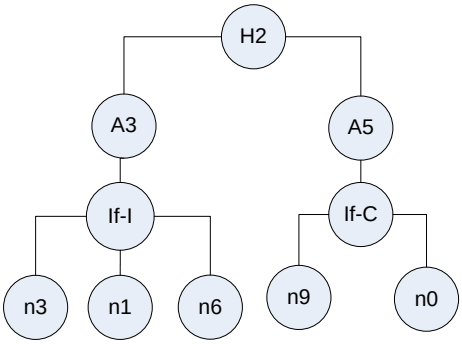

Figure 9.4 Example of if statement

Table 9.2 provides information for the arguments of each element of the function set. In the arity column the comma separation indicates that there are separate operators with different a different arity but the same function.

Table 9.2 Function and terminal arguments for GPHH

Function	**Accepted arguments**	**Description**	**Arity**
M	A, C, I	High-level combiner	1, 2, 3, 4, 5, 6
I	A, C	Iteration	1
A	C, IF-C, IF-I, n	Acceptance method	1
C	C, IF-C, IF-I, n	Low-level combiner	2, 3
IF-C	C, IF-I, IF-C, n	Checks if soft constraint score has not changed	2
IF-I	IF-I, IF-C, C, n	Checks if solution has improved after executing first branch	3
Terminal	Accepted arguments	Description	Arity
n	null	Low-level perturbative heuristic (LPH)	0

9.3 Initial population creation and representation

The initial population is created using the grow method discussed in Chapter 2 section 2.2.2. The grow method cannot add arguments to operators which do not take those operators as

arguments see Algorithm 9.3. Individuals of various depths are generated due to **2b**. A maximum depth is set. Figure 9.5 shows an example of an individual represented as a parse tree. This individual has a depth of 3. It first executes improving or equal move acceptance (A3) and then it checks if the solution has not changed for a number of attempts (IF-C), if it has changed n2 is applied to the candidate solution otherwise n0 is applied. Following this the move acceptance is changed to step counting acceptance (A6) and n11 is applied to the candidate solution.

1. Create a root node
2. For each argument:
    a. **If** maximum depth, select a compatible terminal node
    b. **Else** Select a compatible function or terminal node
3. Repeat 2. For each function node.
4. Once all branches contain a terminal end method

Algorithm 9.3 The grow method

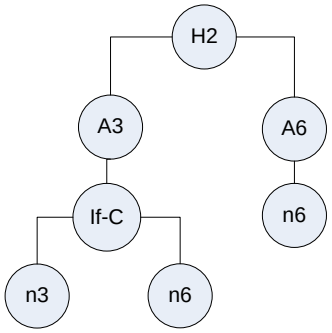

Figure 9.5 Example of an individual or parse tree

9.4 Genetic operators

GPHH uses probability to decide whether to use crossover or other genetic programming operators. Crossover (Figure 9.6) is used for exploration. This is the predominant operator used for genetic programming. The other genetic programming operators consist of: three global search operators namely: Mutation (Figure 9.7), Create and Point mutation (Figure 9.9) and one local search operator: Permutation (Figure 9.8).

Standard genetic programming crossover using strong typing is used. Only compatible types can be exchanged. An example of crossover for GPHH is given in Figure 9.6. In this example the two offspring are created, the better of the two will replace an individual in the population with lower fitness chosen through inverse tournament selection shown in Algorithm 9.4. In this example an acceptance method (A6) node in parent 1 is replaced with function node (C3) from parent 2. This creates an offspring that combines 3 more heuristics. While the second offspring only differs from parent 2 by having a second acceptance method and a new low-level heuristic, on the right branch of H2, this allows for simpler trees to also be considered. Entire sub trees may be swapped providing a point in the second parent's tree contains a compatible argument.

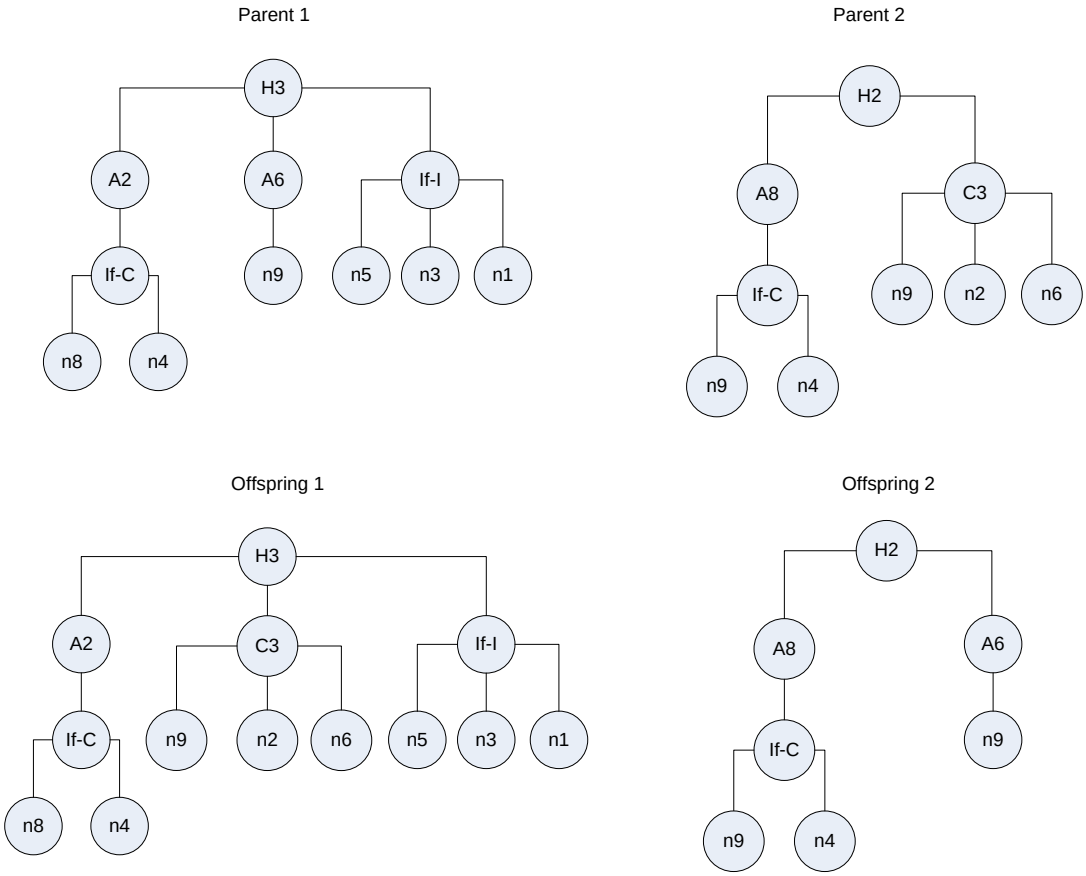

Figure 9.6 GPHH Crossover example

1. Select random individuals from the population equal to the parameter tournament size
2. Set the worst individual as the first individual in the sample
3. Repeat until there are no individuals to compare against:
    a. Compare the worst individual's fitness with the next individual in the sample
    b. **If** the fitness of the worst individual is higher than the compared individual, replace the worst individual with the compared individual.
4. Return the worst individual

Algorithm 9.4 Inverse tournament selection

GPHH mutation, selects a node in the tree at random and then uses the grow method to replace that node with a new subtree. In Figure 9.7, the parent's IF-I node has been replaced with an entirely new subtree starting with IF-C.

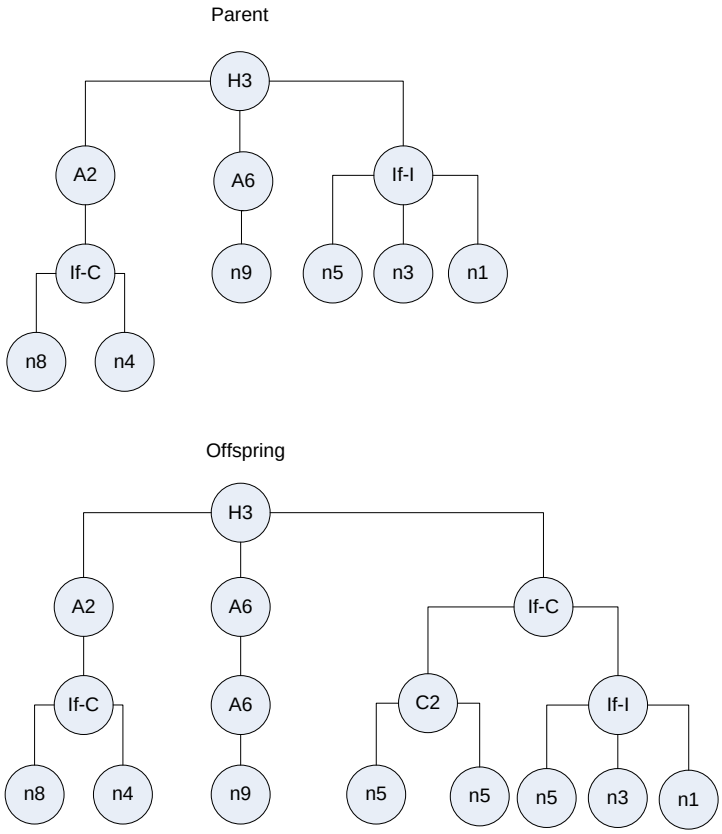

Figure 9.7 GPHH Mutation example

Permutation is an operator that changes the order of the arguments of a selected node. This is done by selecting a random number of changes up to the number of function and terminal set elements in the tree). Then for each change, a node is randomly selected and a swap of the node with another node that is compatible with Table 9.2 is performed. The nodes swapped are removed from the list of nodes. If the maximum depth is exceeded the operator is attempted again until a valid parse tree is created. An example can be seen in Figure 9.8. Where a tree has been completely rearranged but still contains the same individuals as its parent.

Point mutation changes a single terminal node to a different terminal value this can be seen in Figure 9.9 where the terminal node with the label "n0" has been changed to have the label "n9". The terminal node is selected uniformly at random. There are not different types of terminals so point mutation cannot create invalid trees.

The create operator is a more severe form of mutation where instead of selecting a random node, the entire tree is replaced with a newly created individual. This process uses the grow method (Algorithm 9.3).

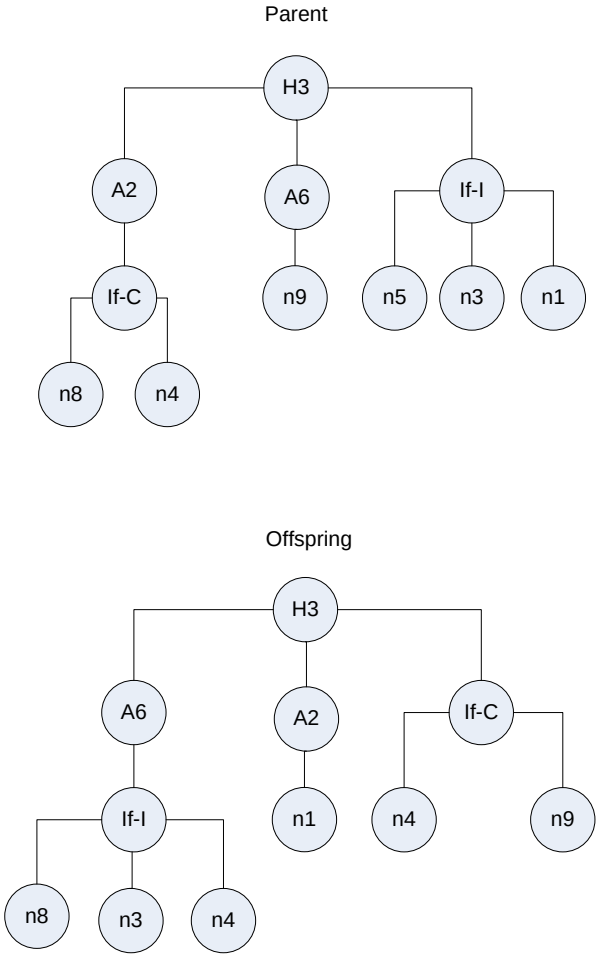

Figure 9.8 GPHH Permutation example

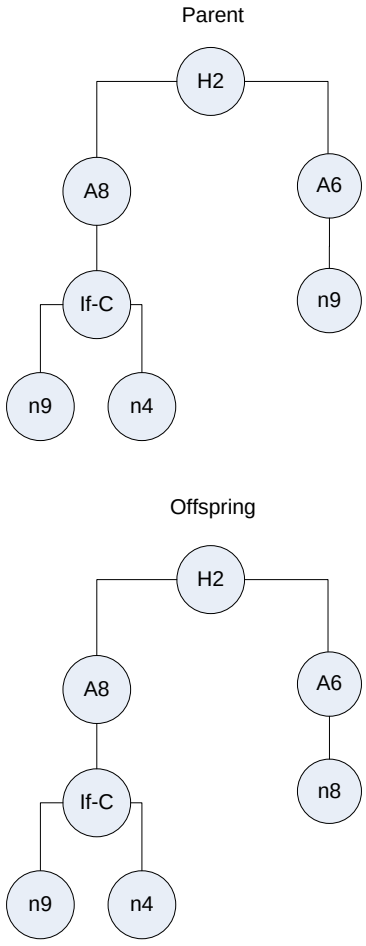

Figure 9.9 GPHH point mutation example

9.5 Parameters

Table 9.3 details the parameters used for GPHH simulations. The following were decided during the study; the fitness function, number of runs for each individual created, the time per run, IF-C limit, tournament size and maximum tree depth, number of generations, the population size, the number of offspring changed by genetic operators at each generation, the probability used for the n9 LPH and the probability of crossover. This was done to explore a number of possible configurations. In this approach if crossover was not selected, one of four exploitation operators were applied to a selected individual. These are: mutation, point mutation, permutation and create.

Each individual is evaluated by 10 runs; each run is limited to half a second, each run works on a randomly generated candidate solution. The individual is executed on the selected instance(s). IF-C is an element of the function set intended to escape local optima if a solution is converging, the limit is set at 30, after 30 consecutive applications to a candidate solution without an improvement in soft constraint score the alternate branch of the IF-C function is taken.

Table 9.3 Parameters used for GPHH runs

Parameter	Value
Fitness function	minimum×average
Runs for individuals	10
Time period for each execution of the evaluation phase Algorithm 9.2	0.5 Seconds
IF-C limit	30
Generations	100
Tournament size	5
Maximum tree depth	12
Population size	100
Offspring per generation	8
Probability Swap	0.96
Crossover	0.85
Benchmark instance data set	The First Nurse Rostering Competition Instances

9.6 Summary

This chapter has presented the model and algorithm for a generative perturbation hyper-heuristic using strongly typed genetic programming. The chapter firstly covers the general algorithms for the approach, and then it gives details of the terminal and function set and representation. Then the initial population generation and genetic operators are covered, finally the chosen parameters are presented.

Chapter 10 Results and discussion

This chapter details the results of the two approaches developed to achieve the objectives outlined in Chapter 7. Section 10.1 presents the results of the genetic algorithm selection perturbative hyper-heuristic (SPHH). Section 10.2 presents the results of the genetic programming generative perturbation hyper-heuristic (GPHH). Section 10.3 presents a comparison of the two approaches results. Section 10.4 gives a comparison to the state of the art for nurse rostering optimization done using the INRC2010 benchmark data set.

10.1 Genetic algorithm selection perturbative hyper-heuristic results (SPHH)

This section presents the results obtained by the approach described in Chapter 8 which dealt with the development of a genetic algorithm selection perturbative hyper-heuristic for solving the nurse rostering problem.

The SPHH was applied to the benchmark instances from the INRC2010 benchmark data set. These will be compared by separating the instances into the three tracks of; sprint, medium and long. SPHH found feasible solutions for all instances from the INRC2010 benchmark set. The results are measured in terms of quality using the number of soft constraint violations. The lowest number of soft constraint violations obtained by SPHH (minSCV) are compared to the best known soft constraint results (BKR). The aim of a hyper-heuristic is not to perform better than state of the art but to perform generally well for a set of problem instances. To this end while good results are desirable they do not necessarily show that the hyper-heuristic is good or bad.

Table 10.1, Table 10.2 and Table 10.3 display the results obtained by SPHH. Table 10.1 for the sprint instances, Table 10.2 for the medium instances and Table 10.3 for the long instances. Each table shows the BKR for each instance, the minSCV obtained over 30 runs of SPHH for each instance, the average number of soft constraint violations (avgSCV) obtained by SPHH over 30 runs for each instance, the average generations taken when a solution was found for each run, the percentage difference between the minSCV obtained by SPHH, the BKR and the standard deviation of the soft constraint violations over 30 runs for each instance.

Table 10.1 SPHH results for sprint instances from INRC2010

Instance	BKR	minSCV	avgSCV	Average generation	Percentage difference	Standard deviation
sprint_early1	56	56	56.1	1.67	0.00%	1.67
sprint_early2	58	58	58.23	2.03	0.00%	2.03
sprint_early3	46	51	51.27	1.9	10.87%	1.9
sprint_early4	59	59	59.47	3.23	0.00%	3.23
sprint_early5	58	58	58.07	1.2	0.00%	1.2
sprint_early6	54	54	54.1	2.23	0.00%	2.23
sprint_early7	56	56	56.47	3.03	0.00%	3.03
sprint_early8	56	56	56.23	1.47	0.00%	1.47
sprint_early9	55	55	55.57	3	0.00%	3
sprint_early10	52	52	52.5	2.57	0.00%	2.57
sprint_late1	37	39	40.43	3.57	5.41%	3.57
sprint_late2	42	43	44.2	3.9	2.38%	3.9
sprint_late3	48	48	50.23	5	0.00%	5
sprint_late4	73	75	82.77	7.43	2.74%	7.43

Instance	BKR	minSCV	avgSCV	Average generation	Percentage difference	Standard deviation
sprint_late5	44	45	45.93	3	2.27%	3
sprint_late6	42	42	42.53	2.6	0.00%	2.6
sprint_late7	42	43	45.47	6.67	2.38%	6.67
sprint_late8	17	17	17.43	1.07	0.00%	1.07
sprint_late9	17	17	17.47	1.23	0.00%	1.23
sprint_late10	43	49	51.73	1	13.95%	1
sprint_hidden1	32	33	35.63	5.17	3.13%	5.17
sprint_hidden2	32	32	34.8	2.6	0.00%	2.6
sprint_hidden3	62	62	64.93	5.47	0.00%	5.47
sprint_hidden4	66	66	67.67	3.87	0.00%	3.87
sprint_hidden5	59	59	61.7	3.67	0.00%	3.67
sprint_hidden6	130	135	152.6	5.03	3.85%	5.03
sprint_hidden7	153	156	178.87	4.1	1.96%	4.1
sprint_hidden8	204	207	228.57	3.13	1.47%	3.13
sprint_hidden9	338	343	357.93	5.63	1.48%	5.63
sprint_hidden10	306	306	323.33	4.87	0.00%	4.87

Table 10.2 SPHH results for medium instances from INRC2010

Instance	BKR	minSCV	avgSCV	Average generation	Percentage difference	Standard deviation
medium_early1	240	240	241.87	4.33	0.00%	4.33
medium_early2	240	240	240.93	7.43	0.00%	7.43
medium_early3	236	237	237.53	5.3	0.42%	5.3
medium_early4	237	237	238.23	6.3	0.00%	6.3
medium_early5	303	303	304.47	8.37	0.00%	8.37
medium_late1	157	165	179.37	12.6	5.10%	12.6
medium_late2	18	24	27.33	13.7	33.33%	13.7
medium_late3	29	31	35.53	8.97	6.90%	8.97
medium_late4	35	37	39.43	11.6	5.71%	11.6
medium_late5	107	127	142.27	14.3	18.69%	14.3
medium_hidden1	121	134	157.93	13.3	10.74%	13.3
medium_hidden2	221	243	269.5	13.8	9.95%	13.8
medium_hidden3	35	36	40.83	11.93	2.86%	11.93
medium_hidden4	78	86	91.43	9.17	10.26%	9.17
medium_hidden5	119	125	141.53	11.67	5.04%	11.67

Table 10.3 SPHH results for long instances from INRC2010

Instance	BKR	minSCV	avgSCV	Average generation	Percentage difference	Standard deviation
long_early1	197	197	197.07	3.6	0.00%	3.6
long_early2	219	220	222.13	10.57	0.46%	10.57
long_early3	240	240	240	1	0.00%	1
long_early4	303	303	303.1	3.87	0.00%	3.87
long_early5	284	284	284.07	3.93	0.00%	3.93
long_late1	235	250	262.5	17.03	6.38%	16.97
long_late2	229	253	263.13	18.87	10.48%	18.86
long_late3	220	256	273.5	16.53	16.36%	16.38
long_late4	221	250	278.03	16.1	13.12%	16.1
long_late5	83	87	92.83	19.73	4.82%	19.79
long_hidden1	346	369	398.43	19.13	6.65%	19.13
long_hidden2	89	90	93.4	16	1.12%	16
long_hidden3	38	42	48	18.97	10.53%	18.97
long_hidden4	22	24	31.87	18	9.09%	18
long_hidden5	41	51	60.3	16.77	24.39%	16.77

SPHH performs well for the sprint instances. For eighteen of the thirty sprint instances the BKR was matched by then minSCV obtained by SPHH. SPHH obtained a minSCV within 5% of the

BKR for nine instances. The average percentage difference of the minSCV to the BKR was 1.73%.

SPHH finds good solutions for the majority of sprint instances. SPHH takes on average 3.38 generations to find good quality solutions for sprint instances. This is the average for all runs of SPHH of the generations it took to find a solution which could not be improved upon.

SPHH performs well for medium instances. For four of the fifteen medium instances the minSCV obtained by SPHH matched the BKR. SPHH obtained a minSCV within 10% of the BKR for eight instances. The average percentage difference of the minSCV to the BKR was 7.27%.

SPHH finds good solutions for the majority of the medium instances. Solutions are found within an average of 10.18 generations. This suggests that SPHH in general finds these harder to solve than the sprint instances.

SPHH performs well on long instances. For four of the fifteen long instances the minSCV obtained by SPHH matched the BKR. SPHH obtained a minSCV within 10% of the BKR for five instances. The average percentage difference of the minSCV to the BKR was 6.89%.

SPHH finds good solutions for a majority of the long instances. Solutions are found within an average of 13.34 generations. This suggests that they were more difficult to solve than the medium instances.

SPHH is good at finding good quality solutions for 65% of the INRC2010 benchmark data set, the minSCV results are within 5% of the BKR. For 81% of all instances the minSCV obtained was within 10% of the BKR. This suggests SPHH obtains results reasonably close to the BKR. SPHH is good at solving the sprint instances which have fewer nurses. The majority of instances where SPHH obtains minSCV results which match the BKR have fewer constraints. These were generally the instances released early for the INRC2010 competition. SPHH still performed well on instances with additional constraints but it is clear these instances were more difficult for SPHH.

10.2 Genetic programming generative perturbation hyper-heuristic results (GPHH)

This section presents the results of using an approach called GPHH to solve the nurse rostering problem. The approach was presented in Chapter 9. This approach has two phases; the first phase is the evolution of a heuristic in the form of a strongly typed parse tree. The second phase is the application of that evolved heuristic applied to the INRC2010 benchmark instances. Section 10.2.1 presents the results of evolving heuristics. Section 10.2.2 presents the results of using the evolved heuristics to solve the nurse rostering problem from the INRC2010 benchmark set.

10.2.1 Evolving low-level perturbation heuristics using GPHH

The genetic programming approach described in Chapter 9 was used to evolve perturbation heuristics. The heuristics were evolved over 30 runs for 100 generations. The run with the lowest fitness was selected as the best evolved heuristic to be applied to the INRC2010 benchmark set. In Table 10.4 it can be seen that the average fitness is generally in the middle of

the maximum and the minimum fitness values. The effectiveness of the evolved perturbation heuristics tested by applying the evolved heuristics to the INRC2010 benchmark instances.

Table 10.4 Results of evolving heuristics

Evolution instance(s)	Min fitness	Max fitness	Average fitness	Standard deviation
S	2197.70	2317.13	2262.496	1511.754
E	53344.30	57489.07	54525.76	4653347.70
H	25737.10	35767.30	30403.00	459912
L	119189.73	203926.33	147860.43	1675204.48
SE	3364.00	3369.80	3365.16	2.36
SH	4666.20	4936.80	4794.64	70.60
SL	1793.40	1822.80	1805.44	6.21
ME	95502.60	98022.00	96381.35	594.56
MH	132224.40	212534.40	165826.2	15855.08
ML	1953.90	3195.90	2344.16	341.82
LE	97605.20	102091.50	99148.68	1084.73
LL	310511.20	563861.60	421691.40	73301.78
LH	18066.80	31990.70	23239.04	3622.99

Table 10.5 shows the best evolved heuristic found. The evolution instance(s) is shown in Chapter 7 section 7.5 in Table 7.1. These were selected by the lowest fitness value found in 30 runs. The average generation for finding the best evolved heuristic was generation 76. The lowest generation of an evolved heuristic was 53 for **SL**. The highest generation for an evolved heuristic was 98 for the instance **LE**.

Table 10.5 Generated Heuristics

Evolved LPH	Generation found	Heuristic
S	87	H1, (, A5, (, IF-C, n9, (, C2, (, C3, n7, n6, (, C3, n9, n9, n9,),), (, IF-C, n2, (, C3, n9, n2, (, C2, (, C2, n10, n10,), (,IF-I, n9, n0, n9,),),),),),),),)
E	71	H1, (, A5, (, IF-C, (, C3, (, C3, (, C3, (, C3, n4, n4, n12,), (,IF-I, n1, n4, n12,), n4,), (, C3, n9, n9, n7,), n2,), (, C3, n8, n9, n5,), n3,), n7, (, C3, n4, n4, n12,),), (, C3, (, C2, n7, n7,), (, C2, n8, n4,), n12,),),)
H	73	H1, (, A7, (, IF-C, (, C3, n9, n9, n12,), (, IF-C, (, C3, n9, n9, n12,), (, IF-C, (, C3, n6, n9, n9,), (, IF-C, (, C3, n9, n11, n12,), (, IF-C, (, C3, n9, n11, n12,), (, IF-C, (, C2, n7, n6,), (,IF-I, n2, n0, n2,),),),),),),),),)
L	95	H1, (, A5, (,IF-I, (, C3, n7, n2, (, C3, (,IF-I, (, C3, n1, n2, (, C3, (, IF-C, (, C2, n1, n2,), n2,), n0, (, IF-C, (, C3, n9, n9, n8,), (, C2, n9, n9,),),),), n1, n2,), n0, (, IF-C, (, C3, n9, n9, n8,), (, C2, n2, n1,),),),), n1, n1,),)
SE	57	H2, (, A3, (,IF-I, (, C3, (, C3, n9, n7, (, C3, n8, n9, n9,),), n12, (, C2, n8, n8,),), n7, (, C2, (, C3, n8, n4, n9,), n2,),),), (, A3, (,IF-I, (, C3, n9, n7, n12,), n9, (, C2, (, C3, n8, n8, n9,), n2,),),)
SH	57	H1, (, A7, (, C3, (, C3, n9, n9, n0,), (, C2, n2, n2,), (, IF-C, (, C3, n2, (, C3, n9, n9, n0,), n8,), n13,),),)
SL	53	H1, (, A5, (, C2, (, C3, n9, n9, n2,), (, C3, n9, n9, n12,),),)
ME	95	H1, (, A3, (, C3, (,IF-I, (, C3, n9, n2, n9,), (, C3, (, IF-C, (, C3, n9, n8, n9,), n7,), n4, n10,), n13,), n4, (, C3, n9, n8, n9,),),)
ML	91	H1, (, A3, (, C3, n4, (, C2, (, C3, n6, (, C2, (, C2, (, C3, n6, (, C2, (, C2, n0, n13,), n9,), (,IF-I, n2, n13, (, C2, (, C3, n0, n8, n13,), n9,),),), n9,), n9,), (,IF-I, n4, n13, (, C2, (, C2, (, C2, (, C2, n0, n10,), n9,), n9,), n9,),),), n9,), (,IF-I, n7, n9, (, C3, n8, n8, n2,),),),)
MH	56	H1, (, A7, (, C2, (, C2, (, C3, (, C2, n9, n9,), n2, (, C3, n8, n1, n9,),), (, IF-C, n9, n13,),), n2,),)
LE	98	H1, (, A7, (, C3, (, C2, (, C2, (, C3, (, C2, n7, (, IF-C, (, C3, n1, n1, n1,), n4,),), n9, (, C2, (, C3, n1, n1, n7,), (, C3, n11, n11, n7,),),), n4,), n4,), n7, (, C3, n1, n2, n9,),),)
LL	71	H1, (, A3, (, C3, (, C3, n10, n8, n4,), (, C3, (, C3, n2, n11, n13,), (,IF-I, n9, n9, n12,), n0,), (,IF-I, (, C3, n2, n11, n10,), n7, (,IF-I, n9, n9, n9,),),),)
LH	88	H2, (, A5, n6,), (, C3, n7, (,IF-I, (, C3, n2, (, C2, n10, n11,), n9,), n6, (, C3, (, C3, n11, n6, n0,), n10, n0,),), (,IF-I, (, C3, n7, (,IF-I, (, C3, n2, (, C2, n10, n11,), n9,), n6, (, C3, (, C3, n11, n6, n0,), n10, n0,),), (,IF-I, (, C3, n2, (, C2, n11, n0,), n9,), n2, (, C3, (, C3, n6, n6, n9,), n10, n0,),),), n2, (, C3, (, C3, n6, n6, n9,), n10, n0,),),)

Table 10.6 shows that the evolved heuristics were the fittest individuals for 23.69 generations on average. This suggests that the 100 generation limit was adequate. It is still possible 100 generations was not enough for the instances that were used to evolve **S**, **L**, **ME**, **ML**, **LE** and **LH** as all were found above 80 generations. It is hypothesized that the aforementioned evolved heuristics are likely to be the poorest performing of the evolved heuristics. The evolved heuristic **SL** only contains three unique LPHs. While it was the best low-level heuristic evolved for that instance, it is hypothesized that it will not perform well when compared to other evolved heuristics with more diversity of heuristics and choice.

Table 10.6 The evolved heuristics and number of generations

	Generations active
Minimum	2
Maximum	47
Average	23.69

10.2.2 Results of applying evolved heuristics

This section presents the results of applying the evolved heuristics to all INRC2010 instances. The evolved heuristics found feasible solutions for all instances. These will be compared by separating the instances into the three tracks: sprint, medium and long. Results are compared to the BKRs in this section and will be compared to SPHH in section 10.3. These evolved heuristics are not expected to perform better than a selection perturbative hyper-heuristic as the intent is to evolve perturbation heuristics.

Table 10.7 shows the minSCV results found for the sprint instances by the evolved heuristics. Table 10.8 shows the avgSCV results found for sprint instances by the evolved heuristics. Table 10.9 shows the percentage difference of the minSCV from the BKR for sprint instances. Table 10.10 shows the averages of the avgSCVs for each evolved heuristic for all sprint instances.

The evolved heuristics found using GPHH on average obtains minSCV values that match the BKRs for 14.76 sprint instances. Three evolved heuristics obtain minSCVs that match 17 BKRs for sprint (**H**, **SE**, **ME**), **LE** obtains minSCVs which match only 9 BKRs for sprint instances. On average GPHH obtains minSCVs for a further eight sprint instances which are within 5% of the BKR. Seven of the evolved heuristics obtained minSCVs which were under 3% on average away from the BKR. **H**, **MH**, **S** have the best performance for obtaining minSCVs for sprint instances. Across sprint instances the evolved heuristics that produced the best avgSCV results were **S**, **H** and **ME**.

Table 10.7 minSCV results for sprint instances

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
sprint_early1	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early2	58	58	58	58	58	58	58	58	58	58	58	58	58	58
sprint_early3	46	51	51	51	51	51	51	51	51	51	51	51	51	51
sprint_early4	59	59	59	59	59	59	59	59	59	59	59	60	59	59
sprint_early5	58	58	58	58	58	58	58	58	58	58	58	58	58	58
sprint_early6	54	54	54	54	54	54	54	54	54	54	54	54	54	54
sprint_early7	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early8	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early9	55	55	55	55	55	55	55	55	55	55	55	56	55	55
sprint_early10	52	52	52	52	52	52	52	52	52	52	52	52	52	52
sprint_late1	37	38	40	38	39	39	39	39	39	38	38	40	39	39
sprint_late2	42	43	43	43	43	42	42	43	43	43	43	45	43	43
sprint_late3	48	49	49	48	49	49	49	50	49	49	49	50	49	50
sprint_late4	73	80	78	79	76	80	80	75	77	78	77	90	77	75

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
sprint_late5	44	45	45	45	45	45	44	45	45	45	45	47	45	45
sprint_late6	42	42	42	42	42	42	42	42	42	42	43	43	42	42
sprint_late7	42	46	47	44	47	44	45	45	44	43	45	49	46	44
sprint_late8	17	17	17	17	17	17	17	17	17	17	17	17	17	17
sprint_late9	17	17	17	17	17	17	17	17	17	17	17	17	17	17
sprint_late10	43	45	48	45	47	46	47	47	46	46	46	50	48	47
sprint_hidden1	32	33	33	33	33	32	33	33	33	33	34	37	34	32
sprint_hidden2	32	32	32	32	32	32	32	33	32	32	32	35	32	32
sprint_hidden3	62	63	64	62	63	63	63	63	62	62	63	64	62	64
sprint_hidden4	66	66	67	66	66	67	67	68	66	66	67	68	68	67
sprint_hidden5	59	60	59	60	59	59	61	60	59	60	60	60	60	59
sprint_hidden6	130	144	172	150	160	166	178	154	163	149	170	160	166	167
sprint_hidden7	153	161	162	156	161	156	157	163	166	161	161	171	158	158
sprint_hidden8	204	211	221	217	215	211	216	213	215	214	220	239	221	215
sprint_hidden9	338	345	348	343	346	340	343	345	339	352	347	361	349	340
sprint_hidden10	306	306	306	306	306	306	312	311	306	306	306	324	318	306

Table 10.8 avgSCV results for sprint

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
sprint_early1	56.57	56.5	56.53	56.63	56.53	57.03	56.9	56.3	56.37	56.8	57.43	56.8	56.87
sprint_early2	58.67	58.6	58.53	58.7	58.23	59.23	59	58.2	58.3	58.8	59.93	58.77	59.17
sprint_early3	52.17	51.93	52	52.37	51.47	52.5	52.43	51.87	51.97	52.3	53.2	52.47	52.5
sprint_early4	60.33	60.43	60.23	60.63	60.07	60.73	60.97	60.1	60.3	60.7	62.23	60.33	61.03
sprint_early5	58.03	58.03	58	58.1	58.03	58.17	58.07	58.03	58.03	58.03	58.7	58	58.17
sprint_early6	54.03	54.13	54.17	54.27	54.03	54.8	54.77	54.1	54.1	54.43	55.3	54.33	54.6
sprint_early7	56.77	57.03	56.8	56.87	56.5	57.63	57.4	56.6	56.83	57.1	57.87	57	57.8
sprint_early8	56.87	56.37	56.47	56.67	56.23	56.8	56.73	56.37	56.53	56.6	56.93	56.53	57
sprint_early9	55.97	56.37	56.1	56.3	56	56.47	56.43	56.1	56	56.17	58.13	56.87	56.83
sprint_early10	52.9	53.33	52.8	53	52.77	53.3	52.93	52.87	52.6	53	54.53	53.17	53.67
sprint_late1	40.63	41.9	40.87	41.07	40.73	41.37	41.47	40.87	40.63	41.43	44.37	41.8	42.1
sprint_late2	45.03	45.53	45.03	45.27	45.2	45.4	45.47	45.2	45.53	45.4	49.07	45.57	46.07
sprint_late3	51.4	51.67	51.37	51.53	50.73	52.3	51.6	51.57	51.13	51.27	55.03	51.83	52
sprint_late4	87.97	89.13	87.17	86.17	87.67	88.77	87.7	85.47	88.8	86.17	104.47	87.47	89
sprint_late5	46.9	47	46.5	46.97	46.73	46.97	46.73	46.87	46.47	47.03	49.37	46.97	47.37
sprint_late6	43.4	43.63	43.03	43.53	43.37	43.6	43.47	43.33	43.13	43.5	45.53	43.7	43.73
sprint_late7	51.63	53.63	51.57	52.2	52.03	51.07	51.17	51.47	50.07	52.33	59.07	52.13	52
sprint_late8	22.8	21.4	20.7	23.03	22	21.73	22.63	21.37	21.13	21.67	26.47	22	21.33
sprint_late9	21.87	23.23	24	21.3	22.43	21.87	22.37	21.97	22.9	23.13	27.7	23.73	23.8
sprint_late10	52.17	53.03	52.47	53.67	52.27	54.33	54.5	53	51.63	53.13	59.23	54.27	55
sprint_hidden1	35.9	37.33	36.4	37.23	35.67	37.07	36.9	36	36.23	36.37	41.3	36.43	37.13
sprint_hidden2	35.1	35.4	34.67	35	34.8	35.53	34.77	34.07	34.47	34.97	39.8	34.77	34.97
sprint_hidden3	66.67	68.17	66.97	68	67.57	67.8	67.63	66.9	67	68	72.83	67.4	68.73
sprint_hidden4	69.27	69.67	69.87	70.17	69.23	69.47	70.03	69.43	69.4	70.1	72.6	70.53	70.7
sprint_hidden5	62.97	63.5	63.03	63.83	62.93	63.7	62.63	62.93	62.77	63.67	67.8	63.23	64.4
sprint_hidden6	161.03	168.8	166.17	167.5	167.07	170.53	164.43	166.57	168.83	170.17	189.07	168	171.5
sprint_hidden7	182.37	191.77	180.37	187.37	183	187.23	182.63	185.63	184	187.43	216.4	191.07	190.67
sprint_hidden8	234.07	240.1	237.13	242.2	238.73	241.2	235.23	240.37	239.5	239.7	272.83	243	241.2
sprint_hidden9	362.9	369.2	363.87	366.83	361.17	365.57	364.37	364.6	365.4	366.53	388.9	367.1	368.1
sprint_hidden10	330.57	340.97	328.53	329.73	337.93	335.77	331.87	331.2	332.7	332.6	375.67	338.13	335.47

Table 10.9 Average percentage away from the BKR for sprint instances

Evolved heuristic	Percentage away from BKR
H	2.20%
MH	2.34%
S	2.40%
SE	2.63%

Evolved heuristic	Percentage away from BKR
ME	2.76%
LL	2.81%
L	2.97%
SL	3.00%
ML	3.33%
SH	3.43%
LH	3.57%
E	3.78%
LE	6.56%

Table 10.10 Average of avgSCV results obtained by evolved heuristics for sprint instances

Evolved heuristic	Average across sprint instances
S	85.57
H	85.71
ME	85.98
SE	86.04
MH	86.09
SL	86.11
L	86.54
ML	86.62
SH	86.93
LH	87.11
E	87.26
LL	87.43
LE	94.39

Table 10.11 shows the minSCV results found for medium instances by the evolved heuristics. Table 10.12 shows the avgSCV results found for medium instances by the evolved heuristics. Table 10.13 shows the percentage difference of the minSCV from the BKR for medium instances. Table 10.14 shows the averages of the avgSCVs for each evolved heuristic for all medium instances.

The evolved heuristics found using GPHH on average find five of fifteen instances within 5% of the BKRs. A further six medium instances are solved within 20% of the BKR. **SL**, **H** and **SE** have the best performance for obtaining minSCVs for medium instances. Across medium instances the evolved heuristics that produced the best avgSCV results were **H**, **SE** and **MH.**

Table 10.11 minSCV results for medium instances

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
medium_early1	240	241	241	240	241	242	243	243	241	241	242	242	242	243
medium_early2	240	241	241	241	242	241	241	241	241	241	241	241	241	242
medium_early3	236	238	238	237	238	237	238	237	238	237	238	238	238	238
medium_early4	237	239	238	238	238	238	239	239	238	238	239	239	238	239
medium_early5	303	304	304	304	304	303	305	304	304	304	304	306	304	305
medium_late1	157	176	182	182	177	180	179	175	182	177	178	201	176	185
medium_late2	18	24	28	24	27	26	29	24	22	25	25	30	27	28
medium_late3	29	35	33	32	33	34	36	34	32	32	36	39	33	36
medium_late4	35	39	39	38	39	38	38	38	37	39	38	45	39	38
medium_late5	107	139	138	130	137	134	140	136	142	139	142	172	135	141
medium_hidden1	121	148	144	148	150	148	156	138	142	146	155	174	149	136
medium_hidden2	221	258	274	263	269	254	278	262	267	271	275	302	270	257
medium_hidden3	35	41	40	37	39	38	39	38	39	39	39	49	37	40

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
medium_hidden4	78	88	86	86	86	87	87	87	85	87	85	91	84	88
medium_hidden5	119	140	138	134	137	139	143	137	137	130	141	146	143	141

Table 10.12 avgSCV results for medium instances

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
medium_early1	243.17	242.87	242.93	243.03	243.23	246.77	244.23	242.77	242.6	243.53	243.9	245.37	244.6
medium_early2	244.37	242.47	242.43	242.67	242.27	245.73	242.8	243.5	242.17	242.83	243.83	244.87	243.97
medium_early3	241.63	238.93	238.9	238.8	239.23	240.37	239.8	239.37	239.03	239.5	240.17	241.6	240.3
medium_early4	242	239.93	239.67	239.77	239.43	240.83	240.6	240.7	239.5	240.5	241	241.9	241.2
medium_early5	308.23	305.73	305.3	305.97	305.37	309.3	306.27	305.53	305.4	306.6	308.93	308.43	307
medium_late1	191.7	200.8	193.4	198.73	193.33	196.67	195.63	192.73	190.33	194.77	218.23	192.5	197.73
medium_late2	31.63	34.77	30.4	31.17	32.27	31.47	30.7	30.53	30.37	32.87	40.3	33.1	34.9
medium_late3	39	39.8	38.13	38.33	39.2	41.33	39.17	41.37	39.03	40.6	47.23	39.63	43.37
medium_late4	42.6	43.3	41.53	43.27	42.5	43.13	41.93	46.4	41.77	43.2	51.33	43.03	44.63
medium_late5	154.07	162.67	152.1	156.7	155	161.87	156.27	173.33	157.23	156.83	195.03	161.1	163
medium_hidden1	166.77	173.17	170.63	173.43	167.9	177.63	169.67	173.9	169.03	172.33	209	177.6	180.43
medium_hidden2	293.87	302.33	288.67	297.2	292.63	300.53	293.2	306.53	293.3	300.4	340.63	296.77	297.17
medium_hidden3	44.37	44.8	43.63	44.63	43	44.2	44.07	43.47	43.97	44.83	53.57	49.77	45.2
medium_hidden4	93.63	95.23	92.07	94.23	93.4	93.83	94.5	93.63	94.3	93.6	103.2	98.67	95.57
medium_hidden5	155.03	165.8	153.97	158.2	154.2	158.9	153.37	151.93	159.5	156.03	187.57	174.87	159.97

Table 10.13 Percentage away from BKR for medium instances

Evolved heuristic	Percentage away from BKR
H	10.74%
ME	10.87%
SL	11.25%
MH	12.05%
SE	12.43%
LH	13.02%
S	13.13%
L	13.42%
E	14.11%
ML	14.16%
LL	14.44%
SH	16.04%
LE	25.48%

Table 10.14 Average of avgSCV results obtained by evolved heuristics for medium instances

Evolved heuristic	Averages
H	164.92
SE	165.53
MH	165.84
S	166.14
SL	166.15
L	167.08
ML	167.23
ME	168.38
SH	168.84
E	168.84
LL	169.27
LH	169.95
LE	181.59

Table 10.15 shows the minSCV results obtained for long instances by each of the evolved heuristics. Table 10.16 shows the avgSCV results found for long instances by the evolved

heuristics. Table 10.17 shows the percentage difference of the minSCV away from the BKR for long instances. Table 10.18 shows the averages of the avgSCVs for each evolved heuristic for all long instances.

The evolved heuristics found using GPHH on average obtain minSCV results for four of fifteen instances within 10% of the BKRs. A further thirteen long instances obtain minSCV results within 20% of the BKRs. **S**, **SL**, **MH** have the best performance for obtaining minSCV results for long instances. Across long instances the evolved heuristics that produced the best avgSCV results were **H**, **MH** and **S.**

Table 10.15 minSCV results for long instances

Instances	BKR	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
long_early1	197	197	197	197	197	197	197	197	197	197	197	197	197	197
long_early2	219	221	221	222	221	221	222	222	221	221	221	223	221	222
long_early3	240	240	240	240	240	240	240	240	240	240	240	240	240	240
long_early4	303	303	303	303	303	303	303	303	303	303	303	303	303	303
long_early5	284	284	284	284	284	284	284	284	284	284	284	284	284	284
long_late1	235	254	262	257	257	255	262	260	257	250	257	283	259	268
long_late2	229	249	273	259	266	263	262	253	258	260	259	305	263	262
long_late3	220	257	268	258	263	258	255	256	262	263	266	294	260	256
long_late4	221	256	271	259	267	264	257	258	259	255	264	299	268	269
long_late5	83	99	104	97	96	96	99	93	95	98	100	123	98	97
long_hidden1	346	368	380	374	374	368	369	370	381	374	377	423	377	374
long_hidden2	89	90	91	89	91	90	90	90	91	90	90	101	91	92
long_hidden3	38	43	45	41	45	43	42	42	40	40	42	55	44	42
long_hidden4	22	23	27	25	24	27	26	27	26	25	27	37	25	28
long_hidden5	41	48	52	48	48	48	45	51	45	46	48	67	48	51

Table 10.16 avgSCV results for long instances

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LE	LH	LL
long_early1	197.13	197.07	197.2	197.47	197.03	198.4	198.2	197	197.2	198.13	197.57	197.87	198.57
long_early2	230.33	223	223.5	223.47	222.57	225.33	224.53	227.27	222.97	224.7	226.53	224.53	225.8
long_early3	240	240	240	240	240	240	240	240.67	240	240	240	241.37	240
long_early4	307.87	303.07	303	303	303.07	303.93	303.8	303.07	303.03	303.67	304.4	306.7	304.37
long_early5	289.27	284.3	284	284.03	284	284.83	284.43	284	284	284.57	285.17	284.27	285.07
long_late1	267.63	285.47	271.57	278.3	274	276.07	273.2	271.67	271.5	276.17	315.87	282.07	282.43
long_late2	273.03	296.27	275.63	285.67	277.43	278.4	276.57	273.1	277.63	282.53	326.77	284.33	289.07
long_late3	275.77	285.3	275.97	279.33	277.3	278.03	277.63	276.2	276.4	279.07	323.2	277.8	284.1
long_late4	273.47	286.47	276.4	283.33	280.3	281.97	275.77	275.43	275.4	277.87	327.6	285	291.47
long_late5	110.7	120.07	111.47	115.47	113.57	116.03	110.9	111.5	110.93	115.7	152.97	118.43	118.07
long_hidden1	396.27	407.57	397.97	401.57	396.43	399.17	399.73	445.53	397.03	399.57	466.3	403.63	414.6
long_hidden2	93.93	98.37	93.2	95.3	94.03	95.4	93.93	105.03	93.97	96.5	111.57	95.9	98
long_hidden3	48	51.8	48.27	50.57	47.8	49.6	47.77	48.47	48.17	50.27	63.07	51.37	50.4
long_hidden4	31.9	34.7	30.87	33.7	31.43	31.77	31.4	30.8	30.47	31.3	49.3	43.77	35.37
long_hidden5	56.27	63.3	55.67	57.1	55.7	58.3	56.2	54.43	56.57	58.4	79.83	58.87	59.87

Table 10.17 Percentage difference from BKRs for long instances

Evolved heuristic	Percentage away from BKR
S	7.46%
MH	7.61%
ME	7.95%
H	8.12%
SH	8.33%
SL	8.89%
SE	9.14%
L	9.16%

Evolved heuristic	Percentage away from BKR
LH	9.41%
ML	9.64%
LL	10.56%
E	12.09%
LE	25.64%

Table 10.18 Average of avgSCV results obtained by evolved heuristics for long instances

Evolved heuristic	Averages
H	205.65
MH	205.68
S	206.10
SL	206.27
SE	206.31
SH	207.82
ML	207.90
L	208.55
ME	209.61
LH	210.39
E	211.78
LL	211.81
LE	231.34

Table 10.19 shows the rankings of the evolved heuristics for the tracks of the INRC2010 benchmark data set for minSCV results obtained by the evolved heuristics. Table 10.20 shows the rankings of the evolved heuristics for the tracks of the INRC2010 benchmark dataset for averages of avgSCV results obtained by the evolved heuristics.

The best evolved heuristics for finding minSCV results were **S**, **H** and **SL**. The best average performance of an evolved heuristic was **H** and the next best were **S** and **MH**. Evolved heuristics created using three seen instances did not generally outperform those evolved using a single seen instance. It should also be noted that evolving using more instances requires more computational time. **H** seems to perform the best with regards to the minSCV results and average avgSCV results. The evolved heuristics of **S**, **MH** and **SE** do have similar performance.

There is no strong relationship between the performance of the evolved heuristic and the constraints which are covered by the instance used for evolution of the algorithm. This is probably due to the design of GPHH which minimizes over fitting by using a randomly initialized candidate solution for each run during evolution (see Chapter 9 section 9.1). Although the seen instances used to evolve **H** covered all constraints, the seen instances used to evolve: **S, SE** and **MH** did not and the heuristics evolved from these instances were comparable to the performance of **H**. **LL** covered all the constraints but had worse performance compared to **H**.

Table 10.19 Ranking of minimum values separated by instance description and type

Rank	INRC2010	Sprint	Medium	Long
1	S	S	SL	S
2	H	H	H	SL
3	SL	SE	SE	MH
4	MH	MH	MH	H
5	SE	LL	ME	SH
6	ME	ME	S	SE
7	L	L	LH	ME
8	LL	SL	L	ML

9	LH	ML	LL	L
10	SH	SH	E	LH
11	ML	E	ML	LL
12	E	LH	SH	E
13	LE	LE	LE	LE

Table 10.20 Average values found ranking separated by instance description and type

Rank	INRC2010	Sprint	Medium	Long
1	H	S	H	H
2	S	H	SE	MH
2	MH	ME	MH	S
4	SE	SE	S	SL
5	SL	MH	SL	SE
6	ML	SL	L	SH
7	L	L	ML	ML
8	ME	ML	ME	L
9	SH	SH	SH	ME
10	LH	LH	E	LH
11	E	E	LL	E
12	LL	LL	LH	LL
13	LE	LE	LE	LE

10.2.2.1 Comparing evolved heuristics to the seen instances used for evolution

The evolved heuristics were applied to the entire INRC2010 benchmark data set. This section aims to see if there was any advantage gained for the evolved heuristics when applied to their seen instances.

Table 10.21 displays the evolved heuristic and the minSCV result obtained for the seen instance used to evolve the heuristic. The table also contains the best minSCV result obtained when applying the other evolved heuristics and which evolved heuristic achieved that result and the difference between both minSCV values. Table 10.22 displays the evolved heuristic and the avgSCV result obtained for the seen instance used to evolve the heuristic. The table also contains the best avgSCV result obtained when applying the other evolved heuristics and the difference between both avgSCV values.

For the seen instance of **SE** and of **LE,** the same minSCV result was obtained by all the evolved heuristics. For the seen instance of **SH,** the same minSCV result was obtained by the evolved heuristics **SH** and **LL**. For the seen instance of **MH**, the evolved heuristics; **H** and **ME** obtained the same minSCV result as **MH**. For the seen instance of **ML,** a 7.6% better minSCV result was obtained by the evolved heuristic, **SE**. For the seen instance of **LH**, a 5.3% better minSCV result obtained by the evolved heuristic **S**.

Table 10.21 Evolved heuristic minSCV performance for seen instance

Evolved heuristic	minSCV	minSCV of best evolved heuristic	Best evolved heuristic	Difference
SE(sprint_early5)	58	58	ALL	0
SH(sprint_hidden4)	75	75	LL	0
SL(sprint_late6)	238	237	H, L, MH	1
ME(medium_early5)	304	303	SE	1
MH(medium_hidden2)	32	32	H, ME	0
ML(medium_late3)	275	254	SE	21
LE (long_early4)	303	303	ALL	0

Evolved heuristic	minSCV	minSCV of best evolved heuristic	Best evolved heuristic	Difference
LH(long_hidden5)	263	249	S	14
LL(long_late2)	51	45	SH	6

Table 10.22 Evolved heuristic avgSCV performance for seen instance

Evolved heuristic	avgSCV	avgSCV of best evolved heuristic	Best evolved heuristic	Difference
SE(sprint_early5)	58.03	58.00	H, LH	0.03
SH(sprint_hidden4)	69.47	69.23	SE	0.24
SL(sprint_late6)	43.47	43.03	H	0.44
ME(medium_early5)	305.53	305.53	H	0.00
MH(medium_hidden2)	39.03	38.13	H	0.90
ML(medium_late3)	300.40	288.66	H	11.74
LE (long_early4)	304.40	303.00	H, L	1.40
LH(long_hidden5)	284.33	273.03	S	11.30
LL(long_late2)	59.87	54.43	ME	5.44

Table 10.23 displays the evolved heuristic and the minSCV result obtained for the seen instances used to evolve the heuristic. The table also contains the minSCV result obtained and the evolved heuristic which achieved that result and the difference to the evolved heuristic for that instance. Table 10.24 displays the evolved heuristic and the avgSCV result obtained for the seen instances used to evolve the heuristic. The table also contains the avgSCV result obtained and the evolved heuristic which achieved that result and the difference to the evolved heuristic for that instance.

The evolved heuristic **H** has been shown to perform well on obtaining minSCV and avgSCV results when compared to the other evolved heuristics performance on the INRC2010 benchmark data set. **H** did not obtain the best minSCV result for the seen instances used to evolve **H**. The same is true for **S** and **H**. The evolved heuristic **L** does find the best minSCV result for one of the instances it was evolved using. As all the evolved heuristics find the same minSCV result for one of the instances used to generate **S**, it would be impossible to say if **S** was tailored to that instance. For avgSCV results only **S** was able to match the average value for one of the instances it was evolved using (sprint_late2). The biggest difference in avgSCV results between the best evolved heuristic and the evolved heuristic using that instance was 4%. The average difference of the avgSCV results was only 1.41%.

Table 10.23 Comparison of evolved heuristics using three seen instances and results obtained for minSCV results of the seen instances in final runs

Evolved heuristic	Instances tested	minSCV	minSCV of best evolved heuristic	Best evolved heuristic	Difference
S	sprint_early2	58	58	ALL	0
	sprint_hidden1	33	32	SE, LL	1
	sprint_late2	43	42	SE, SH	1
E	sprint_early9	55	55	ALL	0
	medium_early4	238	238	H, L, SE, ME, MH, LH	0
	long_early4	303	303	ALL	0
H	sprint_hidden6	150	144	S	6
	medium_hidden5	134	130	MH	4
	long_hidden3	41	40	ME,MH	1
S	sprint_early2	58	58	ALL	0
	sprint_hidden1	33	32	SE, LL	1
	sprint_late2	43	42	SE, SH	1

Table 10.24 Comparison of evolved heuristics using three instances and results obtained for the average of the seen instances in final runs

Evolved heuristic	Instances tested	avgSCV	avgSCV of best evolved heuristic	Best evolved heuristic	Difference
S	sprint_early2	58.67	58.20	ME	0.47
	sprint_hidden1	35.90	35.67	SE	0.23
	sprint_late2	45.03	45.03	S,H	0.00
E	sprint_early9	56.37	55.97	S	0.40
	medium_early4	239.93	239.43	SE	0.50
	long_early4	303.07	303.00	H, L	0.07
H	sprint_hidden6	166.17	161.03	S	5.13
	medium_hidden5	153.97	151.93	ME	2.03
	long_hidden3	48.27	47.77	SL	0.50
L	sprint_late10	53.67	51.63	MH	2.03
	medium_late4	43.27	41.53	H	1.73
	long_late3	279.33	275.77	S	3.57

10.2.2.2 Analyzing the structure of evolved heuristics

It is interesting to look at the structure of the evolved heuristics. In one of the thirteen evolved heuristics the same move acceptance method was used twice. Generally the best evolved heuristics used simulated annealing acceptance (**A7**) or improving or equal acceptance (**A3**), it can be seen that great deluge acceptance **(A5)** did work well for **S**.

Table 10.25 shows the percentage of how the different combinations of the function and terminal sets make up the top four best performing evolved heuristics. Table 10.26 shows the percentage of how the function and terminal sets make up the bottom four worst performing evolved heuristics.

The best move acceptance methods for the nurse rostering problem appear to be simulated annealing (**A7**), improving equal moves (**A3**), great deluge (**A5**). These were used in the best performing evolved heuristics. One evolved heuristic featured two move acceptance methods but was effectively an intron as only the 'improving equal moves' (**A3**) method was used.

Improving only acceptance (**A2**), late acceptance hill climbing (**A4**) or AILLA (**A8**) were not used by any evolved heuristic. Improving only acceptance did not perform well in [213]. It appears the evolution of the heuristics excluded these elements of the function set.

The LPH n1 was not present in any of the best performing evolved heuristics. This suggests that exchanging working shift types between nurses was not an effective way to reduce the soft constraint violations.

H makes use of LPHs that exchange shifts between nurses with different contracts (n11 and n12), **SE** also uses n12. IF-C occurs more in the evolved heuristic **H** and **S,** it also occurs in **LE** and **E**. The heuristic n9 occurs more in **S** than in **H** but occurs in **H** more than in **MH** and **SE**. The heuristic n9 does occur frequently in the poorly performing evolved heuristics. It does not appear that a single function or terminal set element gives an advantage to an evolved heuristic.

Two of the evolved heuristics with the worst performance used the great deluge acceptance method (**A5**). In comparison to the evolved heuristic **S**, it was found that **S** has two LPHs that deal with the day off and shift off requests and features the LPH n9 more frequently than **LL** an

E. These differences allow **S** to perform similar to the best evolved heuristics.

For the worst performing evolved heuristics it can be seen that n1 occurs in two of the four. The heuristic n1 does not occur in any of the best performing evolved heuristics. **LE** only makes use of four unique heuristics and despite using IF-C and IF-I it is unable to make up for its limited moves. The evolved heuristic **LE** also uses n0 which is a blank move low-level heuristic more than any other evolved heuristic. The evolved heuristic **E** has the best diversity of heuristics used but is dependent mostly on n4 where as the best performing evolved heuristics made use of n9 mostly. **E** does not feature n6 or n7. The evolved heuristic **LH** uses n7, n8 and does use n9 the most of the evolved heuristics with poor performance this would suggest it would have performance matching the best evolved heuristics. The evolved heuristic **LH** is similar to **SE** with the exception that in **LH**, NC2 or NC3 functions do not occur, which results in **LH** having less complexity to its tree structure. **LH** only uses the n8 and n7 heuristic once whereas **SE** uses n7 three times and n8 6 times.

IF-C occurs in **H** more than any evolved heuristic. The evolved heuristics **LE, LH** and **LL** use IF-I the most of any evolved heuristic suggesting it is possibly a weaker function generally. **MH** used n9 the most followed by **H** and **SE** yet even occurring frequently in **LH** did not give **LH** notably improved performance on the other worst performing heuristics.

Table 10.25 Best performing heuristics

H		**S**		**MH**		**SE**	
IF-C	17.14%	IF-C	8.00%	IF-I	6.38%	IF-I	5.71%
IF-I	2.86%	IF-I	4.00%	C2	21.28%	C2	8.57%
C2	2.86%	C2	12.00%	C3	10.64%	C3	17.14%
C3	14.29%	C3	12.00%	A3	2.13%	A3	5.71%
A7	2.86%	A5	4.00%	n0	6.38%	n2	5.71%
n0	2.86%	n0	4.00%	n2	4.26%	n4	2.86%
n2	5.71%	n2	8.00%	n4	4.26%	n7	8.57%
n6	5.71%	n6	4.00%	n6	4.26%	n8	17.14%
n7	2.86%	n7	4.00%	n7	2.13%	n9	20.00%
n9	22.86%	n9	28.00%	n8	6.38%	n12	5.71%
n11	5.71%	n10	8.00%	n9	19.15%	H2	2.86%
n12	11.43%	H1	4.00%	n10	2.13%		
				n13	8.51%		
				H1	2.13%		
H1	2.86%						

Table 10.26 Worst performing heuristics

LE		**LL**		**E**		**LH**	
IF-C	4.00%	IF-I	8.00%	IF-C	2.86%	IF-I	13.64%
IF-I	16.00%	C3	26.00%	IF-I	2.86%	A3	4.55%
C2	24.00%	A5	2.00%	C2	5.71%	n0	4.55%
A6	4.00%	n0	14.00%	C3	25.71%	n2	9.09%
n0	24.00%	n2	10.00%	A5	2.86%	n4	4.55%
n1	4.00%	n9	10.00%	n1	2.86%	n7	4.55%
n3	12.00%	n10	12.00%	n2	2.86%	n8	4.55%
n9	8.00%	n11	10.00%	n3	2.86%	n9	22.73%
H1	4.00%	H2	2.00%	n4	20.00%	n12	4.55%
				n5	2.86%	n13	4.55%
				n8	5.71%	H1	4.55%
				n9	8.57%		
				n12	11.43%		
				H1	2.86%		

10.3 Comparison of SPHH and GPHH

This section compares the two approaches developed namely, SPHH a selection perturbative hyper-heuristic and GPHH a generative perturbation hyper-heuristic. For GPHH the best avgSCV results are taken and their corresponding standard deviations. minSCV results from GPHH are the lowest minSCV results from all of the evolved heuristics.

Table 10.27 shows the average minSCV, avgSCV and standard deviation for SPHH and GPHH for sprint instances.

Table 10.27 SPHH Vs. GPHH for sprint instances

	SPHH	GPHH
Average minSCV	79.07	79.53
Average avgSCV	83.41	84.93
Average standard deviation	3.38	2.97

Table 10.28 shows the average minSCV, avgSCV and standard deviation for SPHH and GPHH for medium instances.

Table 10.28 SPHH Vs. GPHH for medium instances

	SPHH	GPHH
Average minSCV	151.00	153.07
Average avgSCV	159.21	164.21
Average standard deviation	10.18	4.68

Table 10.29 shows the average minSCV, avgSCV and standard deviation for SPHH and GPHH for long instances.

Table 10.29 SPHH Vs. GPHH for long instances

	SPHH	GPHH
Average minSCV	194.40	194.13
Average avgSCV	203.22	204.62
Average standard deviation	13.33	5.21

Table 10.30 shows the average minSCV, avgSCV and standard deviation for SPHH and GPHH for all INRC2010 instances. It also presents the results of taking the best average minSCV and avgSCV results from both approaches

Table 10.30 SPHH Vs. GPHH for INRC2010 benchmark set

	SPHH	GPHH	Best of both
Average minSCV	125.88	126.35	125.35
Average avgSCV	132.31	134.67	132.06
Average standard deviation	7.57	3.95	-

SPHH obtains better minSCV and average results across the benchmark data set. SPHH is better at solving the nurse rostering problem. GPHH had lower standard deviations across the benchmark set. This means the application of the evolved heuristics was generally more consistent for obtaining avgSCV results.

Table 10.31 displays the minSCV results for instances where SPHH obtained lower results when compared to results obtained by GPHH.

Table 10.31 Instances where SPHH obtained lower minSCV compared to GPHH

Instances	SPHH	GPHH	Percentage difference
sprint_hidden6	135	144	6.25%
sprint_hidden8	207	211	1.90%
medium_early2	240	241	0.41%
medium_early4	237	238	0.42%
medium_late1	165	175	5.71%
medium_late3	31	32	3.13%
medium_late5	127	130	2.31%
medium_hidden1	134	136	1.47%
medium_hidden2	243	254	4.33%
medium_hidden3	36	37	2.70%
medium_hidden5	125	130	3.85%
long_early2	220	221	0.45%
long_late4	250	255	1.96%
long_late5	87	93	6.45%

Table 10.32 displays the minSCV results for instances where GPHH obtained lower results when compared to results obtained by SPHH.

Table 10.32 Instances where GPHH obtained lower minSCV compared to SPHH

Instances	SPHH	GPHH	Percentage difference
sprint_late1	39	38	2.56%
sprint_late2	43	42	2.33%
sprint_late5	45	44	2.22%
sprint_late10	49	45	8.16%
sprint_hidden1	33	32	3.03%
sprint_hidden9	343	339	1.17%
medium_late2	24	22	8.33%
medium_hidden4	86	84	2.33%
long_late2	253	249	1.58%
long_late3	256	255	0.39%
long_hidden1	369	368	0.27%
long_hidden2	90	89	1.11%
long_hidden3	42	40	4.76%
long_hidden4	24	23	4.17%
long_hidden5	51	45	11.76%

The best minSCV results from GPHH have a slightly higher average percentage difference compared to SPHH. The average percentage difference for instances where SPHH obtains better minSCV results than GPHH is 2.95%. The average percentage difference for instances where GPHH obtains better minSCV results is 3.61%. For 14 instances SPHH obtained minSCV results better than the best minSCV results of GPHH. For 15 instances GPHH obtained minSCV results better than the best minSCV results of SPHH.

Table 10.33 shows how many constraints were present in instances which seemed to favour SPHH and GPHH in terms of minSCV results and those instances which had equal minSCV results. GPHH performed better generally for instances which had the unwanted shift pattern (USP), alternative skill (AS), minimum consecutive working weekends (MinCWW) and No night shift before a free weekend (NNF). SPHH performed better generally on instances with maximum consecutive working weekends (MaxCWW), day off (D) and shift off (S) constraints. The instances with equal minSCV results mostly had the USP, D and S constraints.

Table 10.33 Comparison of constraints of instances with differences in the minimum values obtained

	Shift Types	USP	AS	MinCWW	MaxCWW	NNF	D	S
SPHH favoured instances	4.50	78.57%	35.71%	78.57%	78.57%	64.29%	57.14%	57.14%
GPHH favoured instances	4.40	93.33%	53.33%	100.00%	73.33%	93.33%	46.67%	46.67%
Equal instances	4.10	77.42%	3.23%	45.16%	29.03%	38.71%	90.32%	90.32%

Hypothesis tests were conducted to determine the significance of the results. The hypotheses are:

- H_0: There is no difference in the mean objective value for SPHH and GPHH.
- H_a: The objective value for method SPHH is better than method GPHH.

Table 10.34 shows the Z-value when comparing SPHH against GPHH when compared for all problem instances. It was expected that SPHH would perform better than GPHH because SPHH searches a space of LPHs while GPHH attempts to create new LPHs. This means SPHH performs more optimization in order to solve the nurse rostering problem. SPHH produces statistically significant results when compared to GPHH at the 5% level. Therefore the alternative hypothesis is accepted.

Table 10.34 Statistical test SPHH Vs. GPHH for INRC2010 instances

	SPHH Vs. GPHH
Z-Value	-2.13

Table 10.35 presents a list of z-values for each instance of INRC2010 comparing results of SPHH to those of GPHH.

Table 10.35 GPHH statistically compared to SPHH

Instances	Z-Value	Statistical significance (0 for none, + for over mean, - for under mean)
sprint_early1	0.63	0
sprint_early2	-0.09	0
sprint_early3	0.55	0
sprint_early4	0.99	0
sprint_early5	-0.30	0
sprint_early6	-0.16	0
sprint_early7	0.06	0
sprint_early8	0.00	0
sprint_early9	0.69	0
sprint_early10	0.21	0
sprint_late1	0.29	0
sprint_late2	1.13	0
sprint_late3	0.53	0
sprint_late4	1.77	+
sprint_late5	0.92	0
sprint_late6	1.00	0
sprint_late7	3.34	+
sprint_late8	4.60	+
sprint_late9	17.02	+
sprint_late10	-0.55	0
sprint_hidden1	0.03	0
sprint_hidden2	-1.35	0
sprint_hidden3	1.62	0
sprint_hidden4	2.07	+
sprint_hidden5	1.24	0

Instances	Z-Value	Statistical significance (0 for none, + for over mean, - for under mean)
sprint_hidden6	2.73	+
sprint_hidden7	0.64	0
sprint_hidden8	2.35	+
sprint_hidden9	1.96	+
sprint_hidden10	2.38	+
medium_early1	0.90	0
medium_early2	0.90	0
medium_early3	1.30	0
medium_early4	1.03	0
medium_early5	0.54	0
medium_late1	4.02	+
medium_late2	1.18	0
medium_late3	1.49	0
medium_late4	0.98	0
medium_late5	3.07	+
medium_hidden1	2.71	+
medium_hidden2	5.60	+
medium_hidden3	0.97	0
medium_hidden4	0.36	0
medium_hidden5	4.01	+
long_early1	-0.10	0
long_early2	0.22	0
long_early3	0.00	0
long_early4	-0.14	0
long_early5	-0.09	0
long_late1	1.47	0
long_late2	2.52	+
long_late3	0.53	0
long_late4	-1.39	0
long_late5	4.70	+
long_hidden1	-0.46	0
long_hidden2	-0.07	0
long_hidden3	-0.07	0
long_hidden4	-0.42	0
long_hidden5	-1.81	-

It was found that statistically significant results at the 5% level were found for 16 instances when comparing SPHH to GPHH. These were for two long instances, five medium instances and nine sprint instances. Only the results obtained by GPHH for long_hidden05 were statistically significant at the 5% level of significance, when compared to SPHH. For the majority of instances (43) the null hypothesis is accepted.

GPHH does obtain better minSCV results for certain instances however statistically SPHH is better at minimizing soft constraint violations. One example of a difference SPHH has is the higher standard deviations which were on average 1.9 times greater than the standard deviations of GPHH. This suggests that GPHH has less variability in the results obtained but they are weaker. There is potential that if both methods were combined to take advantage of their respective strengths and weaknesses that results could be further improved.

10.4 Comparison with state of the art

This section will provide an empirical comparison with the state of the art methods which have been applied to the INRC2010 benchmark set. It should be noted that not every study provided results for all the instances in the benchmark data set or statistical means and standard deviations. This comparison only serves as a guideline as such and is provided to give an insight into the performance of the state of the art compared to selection perturbative hyper-heuristic approaches.

Table 10.36 presents the approaches which will be compared with SPHH and GPHH. These results were taken from the studies which provided results and the INRC2010 website [96].

Table 10.36 State of the art competitors for INRC2010 benchmark data set

Description	Label	Reference(s)
Mathematical programming	MP	[100]
Ejection chain and branch and price	ECBP	[57]
Constraint programming	CP	[110]
Hyper-heuristic with greedy shuffle	HHGS	[109]
Adaptive neighbourhood search	ANS	[106]
Stochastic variable neighbourhood search	SVNS	[108]
Integer programming	IP	[101]
Harmony search algorithms	HS	Basic harmony search [115], Global best [116], Greedy Shuffle [118]
Harmony search as a hyper-heuristic	HSHH	[214]

MP was the winner of the INRC2010 competition implemented by Valouxis et al. [100]. Burke and Curtois (ECBP) [57], CP by Nonobe [110], adaptive neighbourhood search (ANS) by Lü and Hao [106] and HHGS by Bilgin et al. [109], ranked within the top 5 for the INRC2010 competition. The integer programming approach (IP) by Santos et al. [101] obtained improvements on the BKRs post-competition, further improvements on a number of BKRs were later obtained by Tassoupoulos et al. [108] (SVNS). There have been three papers looking into harmony search (HS) as a solver for the nurse rostering problem for comparison purposes these results will be combined to take the best minSCV obtained from each HS method. The first was a preliminary implementation of harmony search [115]. The second used an additional approach called global best [116] and another used the greedy shuffle heuristic [118]. Additionally another selection perturbative hyper-heuristic based on harmony search was investigated and results shall be compared with those available (HSHH).

Table 10.37 presents all available minSCV results for their respective methods. Where there is a blank, no data was available.

Table 10.37 Comparison of minSCV results for the state of the art nurse rostering for INRC2010

Instance	SPHH	GPHH	MP	ECBP	CP	HHGS	ANS	SVNS	IP	HS	HSHH
sprint_early1	56	56	56	56	56	57	56	56	56	56	58
sprint_early2	58	58	58	58	58	59	58	58	58	58	60
sprint_early3	51	51	51	51	51	51	51	51	51	46	53
sprint_early4	59	59	59	59	59	60	59	59	59	59	62
sprint_early5	58	58	58	58	58	58	58	58	58	58	58
sprint_early6	54	54	54	54	54	54	54	54	54	54	55
sprint_early7	56	56	56	56	56	56	56	56	56	56	58
sprint_early8	56	56	56	56	56	56	56	56	56	56	56
sprint_early9	55	55	55	55	55	55	55	55	55	55	57
sprint_early10	52	52	52	52	52	52	52	52	52	52	54
sprint_late1	39	38	37	37	37	40	37	37	37	37	
sprint_late2	43	42	42	42	42	44	42	42	42	42	
sprint_late3	48	48	48	48	48	50	48	48	48	48	
sprint_late4	75	75	76	73	76	81	73	73	73	73	
sprint_late5	45	44	44	44	45	45	44	44	44	45	
sprint_late6	42	42	42	42	42	42	42	42	42	42	
sprint_late7	43	43	43	42	43	46	42	42	42	43	
sprint_late8	17	17	17	17	17	17	17	17	17	17	
sprint_late9	17	17	17	17	17	17	17	17	17	17	
sprint_late10	49	45	44	43	44	46	43	43	43	43	
sprint_hidden1	33	32	33				32	32	32	32	
sprint_hidden2	32	32	33				32	32	32	32	
sprint_hidden3	62	62	62				62	62	62	62	
sprint_hidden4	66	66	67				66	66	66	66	
sprint_hidden5	59	59	60				59	59	59	59	
sprint_hidden6	135	144	139				130	130	130	130	
sprint_hidden7	156	156	153				153	153	153	153	
sprint_hidden8	207	211	220				204	204	204	204	
sprint_hidden9	343	339	338				338		338	338	
sprint_hidden10	306	306	306				306	306	306	306	
medium_early1	240	240	240	240	241	242	240	240	240	248	249
medium_early2	240	241	240	240	240	241	240	240	240	248	251
medium_early3	236	237	236	236	236	238	236		236	243	247
medium_early4	237	238	237	237	238	238	237	237	237	245	248
medium_early5	303	303	303	303	304	304	303	303	303	311	315
medium_late1	163	175	159	158	176	163	164		157	175	
medium_late2	22	22	20	18	19	21	20		18	31	
medium_late3	30	32	30	29	30	32	30		29	40	
medium_late4	37	37	36	35	37	38	36	35	35	44	
medium_late5	125	130	113	107	125	122	107		107	137	
medium_hidden1	134	136	131		130		122	121	122	190	
medium_hidden2	257	254	221				224		221	264	
medium_hidden3	35	37	38		36		35	35	36	59	
medium_hidden4	85	84	80				80	79	78	96	
medium_hidden5	125	130	122				120		119	190	
long_early1	197	197	197	197	197	197	197	197	197	215	214
long_early2	220	221	219	219	224	220	222	219	219	248	245
long_early3	240	240	240	240	240	240	240	240	240	246	248
long_early4	303	303	303	303	303	303	303	303	303	314	317
long_early5	284	284	284	284	284	284	284	284	284	296	298
long_late1	250	250	239	235	267	241	237		235	258	
long_late2	253	249	231	229	245	245	229		229	259	
long_late3	253	255	222	220	254	233	222		220	268	
long_late4	262	255	228	221	260	246	227		221	272	
long_late5	89	93	83	83	93	87	83		83	112	
long_hidden1	369	368	363				346		346	417	
long_hidden2	90	89	106			90	89	89	89	106	

Instance	SPHH	GPHH	MP	ECBP	CP	HHGS	ANS	SVNS	IP	HS	HSHH
long_hidden3	42	40	38				38	38	38	49	
long_hidden4	27	23	22				22	22	22	32	
long_hidden5	51	45	41				45	41	41	56	

Table 10.38 shows the percentage of instances where SPHH has higher minSCV results, lower minSCV results and equal minSCV results.

Table 10.38 Summary of SPHH compared to the state of the art for INRC2010 instances

	MP	ECBP	CP	HHGS	ANS	SVNS	IP	HS	HSHH
SPHH >	40.00%	42.50%	16.67%	19.51%	46.67%	39.13%	51.67%	16.67%	0.00%
SPHH <	13.33%	0.00%	23.81%	36.59%	3.33%	0.00%	1.67%	50.00%	90.00%
SPHH =	46.67%	57.50%	59.52%	43.90%	50.00%	60.87%	46.67%	33.33%	10.00%

Table 10.39 shows the percentage of instances where GPHH has higher minSCV results, lower minSCV results and equal minSCV results.

Table 10.39 Summary of GPHH compared to the state of the art for INRC2010 instances

	MP	ECBP	CP	HHGS	ANS	SVNS	IP	HS	HSHH
GPHH >	45.00%	45.00%	26.19%	21.95%	48.33%	39.13%	51.67%	13.33%	0.00%
GPHH <	13.33%	0.00%	19.05%	36.59%	1.67%	0.00%	0.00%	50.00%	90.00%
GPHH =	41.67%	55.00%	54.76%	41.46%	50.00%	60.87%	48.33%	36.67%	10.00%

Table 10.40 presents the percentage of instances where the minSCV results were better or equal to the state of the art.

Table 10.40 Percentage of instances where SPHH and GPHH were better or equal to state of the art

	MP	ECBP	CP	HHGS	ANS	SVNS	IP	HS	HSHH
SPHH	60.00%	57.50%	83.33%	80.49%	53.33%	60.87%	48.33%	83.33%	100.00%
GPHH	55.00%	55.00%	73.81%	78.05%	51.67%	60.87%	48.33%	86.67%	100.00%

SPHH and GPHH were better or equal to the majority of the available minSCV results, with the exception of IP. MP, ECBP, CP, ANS and IP have minSCV results which are lower than SPHH and GPHH for more instances. SPHH and GPHH obtained minSCV results which were better than those obtained by MP, CP, HHGS, ANS, HS and HSHH for at least one instance. On average SPHH and GPHH found better results for 28% of minSCV results obtained by state of the art approaches excluding HSHH, where SPHH and GPHH obtained better or equal minSCV results. Although SPHH and GPHH do not always produce the best results, this empirical comparison does show that the performance of the two approaches is comparative.

Table 10.41 which presents the difference of the sum of available minSCV results obtained by the state of the art and compares it to GPHH and SPHH.

Table 10.41 Difference of average minSCV results for available results for comparing the state of the art to SPHH and GPHH

	SPHH	GPHH
MP	-2.88	-3.05
ECBP	-2.72	-2.90
HHGS	-0.60	-0.77
HS	7.22	7.05
CP	0.45	0.1
ANS	-4.18	-4.35
SVNS	-1.46	-1.41
IP	-4.73	-4.9
HHHS	7.4	7.2

GPHH and SPHH obtained better minSCV results compared to HHHS and HS. GPHH and SPHH had minSCV results which were close to CP, HHGS and SVNS. GPHH and SPHH minSCV results were worse overall when compared to MP, ECBP, ANS and IP. The closeness of the available results of CP, HHGS and SVNS suggests the approaches are still comparable to the state of the art. The other population based approaches (HS and HSHH) do not perform well in general and the available minSCV results are worse than those obtained by GPHH and SPHH. As both SPHH and GPHH obtain similar results to the selection perturbative hyper-heuristic (HHGS) it suggests these are both comparable to other selection perturbative hyper-heuristic approaches.

10.5 Summary

This chapter presents the results of the two developed approaches, SPHH and GPHH. A comparison of SPHH and GPHH is given and an empirical comparison between the state of the art approaches and the two developed approaches is given.

Chapter 11 Conclusions and future work

This chapter presents the overall conclusions based on the research findings of the dissertation. The outcomes are presented with respect to the two objectives outlined in Chapter 1; these are presented in section 11.1. Finally directions for future work building upon this dissertation are presented in section 11.2.

11.1 Objectives and conclusions

- Objective 1: Investigate a genetic algorithm selection perturbative hyper-heuristic for the nurse rostering problem. The approach implemented should be influenced by relevant literature.

A genetic algorithm hyper-heuristic was implemented based on a critical analysis of literature on previous genetic algorithm selection perturbative hyper-heuristics and selection perturbative hyper-heuristics and heuristic search approaches applied to the nurse rostering problem. The developed approach, SPHH performed well on the INRC2010 benchmark set. SPHH found feasible solutions for all instances. SPHH obtained results which were close to those obtained by state of the art approaches. The performance of SPHH was comparative with to an existing selection perturbative hyper-heuristic approach.

- Objective 2: Develop and analyse a genetic programming generative perturbation hyper-heuristic for solving the nurse rostering problem. This approach should create perturbation heuristics that can be used to solve the nurse rostering problem.

A generative perturbative hyper-heuristic was implemented. This generative perturbative hyper-heuristic was created through critical analysis of recent work in literature for evolving perturbation heuristics. Many studies which evolve heuristics used grammatical evolution and this lead to the use of a steady state control model which was beneficial for GPHH. GPHH evolved heuristics which produced feasible solutions for all problem instances in the INRC2010 benchmark set. Results obtained were comparative with the state of the art and GPHH obtained fewer soft constraint violations for some instances in comparison to SPHH. Results suggest that the evolution process may be useful in identifying good function and terminal set elements. The heuristics evolved using more than one seen instance did not necessarily result in better performing evolved heuristics compared to those evolved using only a single seen instance. None of the evolved heuristics were seen to suffer from over fitting. GPHH evolved heuristics with structures that are similar to how human designed meta-heuristics are structured with a single move acceptance method.

- Objective 3: Compare the performance of the two perturbative hyper-heuristics for the nurse rostering problem.

The performance of the two developed approaches namely, SPHH and GPHH were compared using minSCV results, avgSCV results and hypothesis testing. Hypothesis testing was used to determine the statistical significance of the results. Across all instances the results obtained by SPHH were statistically significant compared to those obtained by GPHH. In terms of

individual instances SPHH obtained statistically significant results for 16 instances and GPHH obtained one statistically significant result, this was for the instance of long_hidden05. GPHH was still able to obtain fewer soft constraint violations on a number of instances compared to SPHH. GPHH still showed good performance given that the aim was not to compete with SPHH but to create new perturbation heuristics. GPHH evolved heuristics which performed well on instances individually, if these were used by SPHH system results obtained would probably improve further. SPHH was designed to solve the nurse rostering problem by searching the low-level heuristic space. GPHH was designed to evolve heuristics which were then used to solve the nurse rostering problem but should ideally be used by a meta-heuristic as a neighbourhood operator or as an LPH used by a selection perturbative hyper-heuristic.

11.2 Future work

Based on the results of this research of hyper-heuristics and the nurse rostering problem potential future work will be discussed. Future extensions of the research presented in this dissertation include:

11.2.1 Combining evolutionary selection and generation hyper-heuristics

This would entail using GPHH to evolve heuristics which will be used by a selection perturbative hyper-heuristic such as SPHH to improve an initial candidate solution. This would address the issue of choosing a set of low-level heuristics however a broader and refined function set to GPHH would need to be included.

11.2.2 Coevolving the algorithm parameters for selection and generative perturbation hyper-heuristics

The field of hyper-heuristics is rapidly moving towards hyper-heuristics which generate and design the hyper-heuristic itself. It is still however cumbersome to tune parameters be it through empirical testing or using a tool such as ParamILS [231]. A parallel genetic algorithm or a tuning mechanism built into the selection or generative perturbative hyper-heuristic based on parameter design approaches such as the Taguchi orthogonal arrays[232] could provide a novel solution to parameter tuning. Coevolving parameters for a selection or generative hyper-heuristic would be compared using the standard approaches used by researchers in the field and tools like F-Race[233] and ParamILS.

11.2.3 Generative construction hyper-heuristic for the nurse rostering problem

There are no equivalent construction heuristics to the set of “graph colouring heuristics” for timetabling, for the nurse rostering problem. A genetic programming approach will be implemented to break down what human schedulers attempt to do into components that can be used to evolve low-level construction heuristics for the nurse rostering problem.

11.3 Summary

This chapter gives a summary of the findings of the research of this dissertation and the outcomes of the objectives and how they were fulfilled. Finally, future work based on the observations made during this research is presented.

Bibliography

[1] C. Darwin, “On the origins of species by means of natural selection,” *London: Murray*, 1859.

[2] J. Holland and J. Reitman, “Cognitive systems based on adaptive algorithms,” *ACM SIGART*, vol. 63, no. Bulletin, pp. 49–49, 1977.

[3] D. Goldberg and J. Holland, “Genetic algorithms and machine learning,” *Mach. Learn.*, vol. 2, no. 3, pp. 95–99, 1988.

[4] J. Holland, “Genetic Algorithms,” *Sci. Am.*, vol. 267, no. 1, pp. 337–370, 2011.

[5] J. Koza, Genetic programming: on the programming of computers by means of natural selection, 1st ed. MIT press, 1992.

[6] W. Banzhaf *et al.*, *Genetic Programming, an Introduction*. San Francisco: Morgan Kaufmann Publishers, 1998.

[7] F. Koza, John R. and Bennett, Forrest H and Andre, David and Keane, Martin A. and Dunlap, J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap, “Automated synthesis of analog electrical circuits by means of genetic programming,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 2, pp. 109–128, 1997.

[8] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler, “Co-evolving soccer softbot team coordination with genetic programming,” in *RoboCup-97: Robot soccer world cup I*, Springer, 1998, pp. 398–411.

[9] A. Freitas, “A genetic programming framework for two data mining tasks: classification and generalized rule induction,” *Genet. Program.*, pp. 96–101, 1997.

[10] A. S. Fukunaga, “Evolving local search heuristics for SAT using genetic programming,” in *Genetic and Evolutionary Computation Conference*, 2004, pp. 483–494.

[11] G. Burke, Edmund K and Hyde, Matthew R and Kendall, *Evolving Bin Packing Heuristics with Genetic Programming*, Parallel P. Springer, 2006, pp. 860–869.

[12] R. Poli, W. B. Langdon, N. F. McPhee, J. R. Koza, and J. R. Poli, Riccardo and Langdon, William B and McPhee, Nicholas F and Koza, *A field guide to genetic programming*, no. March. Lulu.com, 2008, p. 250.

[13] T. Blickle and L. Thiele, “A comparison of selection schemes used in genetic algorithms,” *TIK-Report*, 1995.

[14] L. M. Brad, D. E. Goldberg, B. L. Miller, and D. E. Goldberg, “Genetic algorithms, tournament selection, and the effects of noise,” *Complex Syst.*, vol. 9, no. 3, pp. 193–212, 1995.

[15] D. E. Golberg and D. E. Goldberg, “Genetic algorithms in search, optimization, and machine learning,” *Addion wesley*, vol. 1989, p. 102, Oct. 1989.

[16] T. Bäck and T. Back, “Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms,” *Oxford Univ. Press*, 1996.

[17] D. Andre, F. B. III, J. J. R. Koza, F. H. Bennett III, and J. J. R. Koza, “Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem,” in … *Conference on Genetic Programming*, 1996, pp. 3–11.

[18] K. E. Kinnear, “Evolving a sort: Lessons in genetic programming,” *Neural Networks, 1993., IEEE Int. …*, vol. IEEE Inter, no. Neural Networks, pp. 881–888, 1993.

[19] N. Pillay, “An Investigation into the Use of Genetic Programming for the Induction of Novice Procedural Programming Solution Algorithms in Intelligent Programming Tutors,” University of Natal, 2004.

[20] W. S. Bruce, “The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs,” 1995.

[21] C. W. Reynolds, “An evolved, vision-based model of obstacle avoidance behavior,” *From Anim. to Animat.*, vol. 2, pp. 384–392, 1993.

[22] D. J. D. Montana, “Strongly typed genetic programming,” *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995.

[23] D. Dracopoulos and S. Kent, “Genetic programming for prediction and control,” *Neural Comput. Appl.*, pp. 214–228, 1997.

[24] G. Syswerda, “A study of reproduction in generational and steady state genetic algorithms,” *Found. Genet. algorithms*, vol. 2, pp. 94–101, 1991.

[25] C. Ryan, M. O’Neill, and M. O’Neill, “Grammatical evolution: A steady state approach,” *Late Break. Pap. Genet. Program.*, vol. 1, pp. 180–185, 1998.

[26] K. Chellapilla, “Evolving computer programs without subtree crossover,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 3, pp. 209–216, 1997.

[27] S. Luke and L. Spector, “A comparison of crossover and mutation in genetic programming,” *Genet. Program.*, vol. 97, pp. 240–248, 1997.

[28] P. L. Naga *et al.*, “Genetic algorithms for the travelling salesman problem: A review of representations and operators,” *Artif. Intell. Rev.*, vol. 13, no. 2, pp. 129–170, 1999.

[29] S. Africa, “Using Genetic Algorithms to Solve the South African School Timetabling Problem Rushil Raghavjee Nelishia Pillay,” pp. 286–292, 2010.

[30] R. Raghavjee, “A Study of Genetic Algorithms for Solving the School Timetabling Problem,” School of Computer Science, University of Kwazulu-Natal, Pietermaritzburg, 2013.

[31] P. Cowling, G. Kendall, and L. Han, “An Investigation of a Hyperheuristic Genetic Algorithm Applied to a Trainer Scheduling Problem,” in *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC’02.*, 1999, pp. 1185–1190.

[32] D. Corne and R. Ogden, “Evolutionary optimisation of methodist preaching timetables,” *Int. Conf. Pract. Theory Autom. Timetabling*, 1997.

[33] T. Abdelmaguid, “Representations in genetic algorithm for the job shop scheduling problem: A computational study,” *J. Softw. Eng. Appl.*, vol. 3, no. 12, p. 1155, 2010.

[34] R. Raghavjee and N. Pillay, “A Genetic Algorithm Selection Perturbative Hyper-Heuristic for Solving the School Timetabling Problem,” *Orion*, vol. 3, no. 1, pp. 39–60, 2015.

[35] D. Goldberg, B. Korb, and K. Deb, “Messy genetic algorithms: Motivation, analysis, and first results,” *Complex Syst.*, vol. 1, no. 3, pp. 493–530, 1989.

[36] M. Melanie, “An introduction to genetic algorithms,” *Cambridge, Massachusetts London, England, Fifth Print.*, vol. 3, pp. 62–75, 1999.

[37] L. Davis, *Handbook of genetic algorithms*. Van Nostrand Reinhold, New York, 1991.

[38] H. Stringer and A. S. Wu, “Variable-Length Genetic Algorithms and an Analysis of Changes in Chromosome Length Absent Selection Pressure,” 2005.

[39] K. Lindgren, “Evolutionary phenomena in simple dynamics,” in *Artificial life II*, 1992, pp. 295–312.

[40] L. Han and G. Kendall, “An Investigation of a Tabu Assisted Hyper-Heuristic Genetic Algorithm,” in *Evolutionary Computation, 2003. CEC’03. The 2003 Congress on*, 2003, vol. 3, pp. 2230–2237.

[41] G. Nemhauser and L. Wolsey, *Integer and combinatorial optimization*. 1988.

[42] L. Wolsey, “Integer programming,” *Ser. Discret. Math. Optim.*, pp. 113–129, 1998.

[43] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[44] J. Tomlin, “Technical Note—An Improved Branch-and-Bound Method for Integer Programming,” *Oper. Res.*, vol. 19, no. 4, pp. 1070–1075, 1971.

[45] A. Lokketangen and F. Glover, “Solving zero-one mixed integer programming problems using tabu search,” *Eur. J. Oper. Res.*, vol. 106, no. 2, pp. 624–658, 1998.

[46] B. M. W. Cheng, J. H. M. Lee, J. C. K. Wu, and H. Kong, “A Nurse Rostering System Using Constraint Programming and Redundant Modeling,” vol. Informatio, no. IEEE Transactions on 1.1 (1997), pp. 44–54, 1997.

[47] K. Apt and P. Shaw, “Principles of constraint programming,” in *International Conference on Principles and Practice of Constraint Programming*, 2003, pp. 417–431.

[48] P. Van Beek, X. Chen, P. Van Beek, and X. Chen, “CPlan: A constraint programming approach to planning,” in *AAAI/IAAI*, 1999, pp. 585–590.

[49] P. Shaw, “Using constraint programming and local search methods to solve vehicle routing problems,” in *International Conference on Principles and Practice of Constraint Programming*, 1998, pp. 417–431.

[50] R. Soto, B. Crawford, E. Monfroy, W. Palma, and F. Paredes, “Nurse and paramedic rostering with constraint programming: A case study,” *Rom. J. Inf. Sci. Technol.*, vol. 16, no. 1, pp. 52–64, 2013.

[51] A. Land and A. Doig, “An automatic method of solving discrete programming problems,” *Econom. J. Econom. Soc.*, vol. July, no. 1, pp. 497–520, 1960.

[52] E. Lawler and D. Wood, “Branch-and-bound methods: A survey,” *Oper. Res.*, 1966.

[53] R. Horst and H. Romeijn, *Handbook of global optimization volume 2*. 2002, p. 58.

[54] J. Paulavičius, R. and Žilinskas, R. Paulavičius, and J. Žilinskas, “Lipschitz Optimization with Different Bounds over Simplices,” *Simplicial Glob. Optim.*, no. Springer New York, pp. 21–60, 2014.

[55] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, “Branch-and-price: Column generation for solving huge integer programs,” *Oper. Res.*, vol. 46, no. 3, pp. 316–329, 1998.

[56] M. Savelsbergh, “A branch-and-price algorithm for the generalized assignment problem,” *Oper. Res.*, vol. 45, no. 6, pp. 831–841, 1997.

[57] E. K. E. Burke and T. Curtois, “New computational results for nurse rostering benchmark instances,” *Tech. Rep.*, vol. 10, p. 13, 2011.

[58] D. Pinha and Q. P. Q. Zheng, “Branch and Price,” 2012. [Online]. Available: http://www.iems.ucf.edu/qzheng/grpmbr/seminar/Denis_Branch_and_Price.pdf. [Accessed: 10-Dec-2014].

[59] M. Fischetti, J. J. Salazar González, and P. Toth, “A branch-and-cut algorithm for the symmetric generalized traveling salesman problem,” *Oper. Res.*, vol. 45, no. 3, pp. 378–394, 1997.

[60] A. Mathematics, S. Review, M. Padberg, and G. Rinaldi, “A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems,” *SIAM Rev.*, vol. 33, no. 1, pp. 60–100, 1991.

[61] I. I. H. Osman and G. Laporte, “Metaheuristics: A bibliography,” *Ann. Oper. Res.*, vol. 63, no. 5, pp. 511–623, 1996.

[62] F. Glover and G. Kochenberger, *Handbook of metaheuristics*. 2003.

[63] G. Croes, “A method for solving traveling-salesman problems,” *Oper. Res.*, 1958.

[64] L. Cooper and D. Steinberg, “Introduction to methods of optimization,” 1970.

[65] J. J. Grefenstette, “Optimization of control parameters for genetic algorithms,” *IEEE Trans. Syst. Man. Cybern.*, vol. 16, no. 1, pp. 122–128, 1986.

[66] F. Glover, “Tabu search – part I,” *ORSA J. Comput.*, vol. 1, no. 3, pp. 190–206, 1989.

[67] F. Glover, “Tabu search—part II,” *ORSA J. Comput.*, vol. 2, no. 1, pp. 4–32, 1990.

[68] S. Kirkpatrick, “Optimization by simulated annealing: Quantitative studies,” *J. Stat. Phys.*, vol. 34, no. 5, pp. 975–986, 1984.

[69] P. J. M. Van Laarhoven, E. E. H. L. Aarts, P. Van Laarhoven, and E. E. H. L. Aarts, *Simulated annealing*. Springer, 1987, pp. 7–15.

[70] G. Dueck, “New optimization heuristics: The great deluge algorithm and the record-to-record travel,” *J. Comput. Phys.*, vol. 104, no. 1, pp. 86–92, 1993.

[71] N. Mladenović, P. Hansen, N. Mladenović, and P. Hansen, “Variable neighborhood search,” *Comput. Oper. Res.*, vol. 146, no. International Series in Operations Research & Management Science, pp. 61–86, 1997.

[72] Z. Z. W. Geem, J. H. J. Kim, and G. V Loganathan, “A new heuristic optimization algorithm: harmony search,” *Simulation*, vol. 76, no. 2, pp. 60–68, 2001.

[73] D. Weyland, “A rigorous analysis of the harmony search algorithm: How the research community can be misled by a ‘novel’ methodology,” *Model. Anal. Appl. Metaheuristic Comput. Adv. Trends Adv. Trends*, p. 72, 2012.

[74] M. Mahdavi, M. Fesanghary, and E. Damangir, “An improved harmony search algorithm for solving optimization problems,” *Appl. Math. Comput.*, vol. 188, no. 2, pp. 1567–1579, 2007.

[75] M. Padberg, “Harmony Search Algorithms for binary optimization problems,” in *Operations Research Proceedings 2011*, 2012, pp. 343–348.

[76] H. Frøyseth, M. Stølevik, and A. Riise, “A heuristic approach for solving real world nurse rostering problems,” in *The 7th international conference on the practice and theory of automated timetabling*, 2008, p. 5.

[77] E. Burke, P. De Causmaecker, P. De Causmaecker, G. Vanden Berghe, and H. Van Landeghem, “The State of the Art of Nurse Rostering,” *J. Sched.*, vol. 7, no. 6, pp. 441–499, Nov. 2004.

[78] B. Cheang, H. Li, A. Lim, and B. Rodrigues, “Nurse rostering problems—a bibliographic survey,” *Eur. J. Oper. Res.*, vol. 151, no. 3, pp. 447–460, Dec. 2003.

[79] S. Haspeslagh *et al.*, “The first international nurse rostering competition 2010,” *Ann. Oper. Res.*, vol. 218, no. 1, pp. 1–31, Jan. 2012.

[80] T. Curtois, “Novel heuristic and metaheuristic approaches to the automated scheduling of healthcare personnel,” 2007.

[81] R. Karp, *Reducibility among combinatorial problems*, Complexity. Springer, 1972, pp. 85–103.

[82] J. Tien and A. Kamiyama, “On manpower scheduling algorithms,” *Siam Rev.*, vol. 24, no. 3, pp. 275–287, 1982.

[83] A. Wren, “Scheduling, timetabling and rostering—a special relationship?,” *Pract. theory Autom. timetabling*, 1996.

[84] H. E. H. Miller, W. P. W. Pierskalla, and G. J. G. Rath, “Nurse scheduling using mathematical programming,” *Oper. Res.*, vol. 24, no. 5, pp. 857–870, 1976.

[85] M. Warner, B. Keller, and S. Martel, “Automated nurse scheduling.,” *J. Soc. Health Syst.*, vol. 2, no. 2, pp. 66–80, 1990.

[86] D. L. Kellogg and S. Walczak, “Nurse Scheduling: From Academia to Implementation or Not?,” *Interfaces (Providence).*, vol. 37, no. 4, pp. 355–369, Jul. 2007.

[87] M. Miller, “Implmenting Self-Scheduling,” *J. Nurs. Adm.*, vol. 14, no. 3, pp. 33–36, 1984.

[88] R. Hung, “Improving productivity and quality through workforce scheduling,” *Ind. Manag. THEN ATLANTA*, vol. 34, pp. 4–4, 1992.

[89] R. Silvestro and C. Silvestro, “An evaluation of nurse rostering practices in the National Health Service,” *J. Adv. Nurs.*, vol. 32, no. 3, pp. 525–535, 2000.

[90] I. Berrada, J. A. Ferland, and P. Michelon, “A multi-objective approach to nurse scheduling with both hard and soft constraints,” *Socioecon. Plann. Sci.*, vol. 30, no. 3, pp. 183–193, 1996.

[91] E. K. E. Burke, J. Li, and R. Qu, “A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems,” *Eur. J. Oper. Res.*, vol. 203, no. 2, pp. 484–493, 2010.

[92] T. Curtois, “Nurse rostering benchmark instances.” [Online]. Available: http://www.cs.nott.ac.uk/~tec/NRP/.

[93] B. Vanhoucke, Mario, Maenhout, “NSPLib.” [Online]. Available: http://www.projectmanagement.ugent.be/?q=research/personnel_scheduling/nsp.

[94] M. Vanhoucke and B. Maenhout, “NSPLib – A Nurse Scheduling Problem Library : A tool to evaluate (meta-) heuristic procedures,” pp. 1–11, 2005.

[95] S. Petrovic and G. Vanden Berghe, “A comparison of two approaches to nurse rostering problems,” *Ann. Oper. Res.*, vol. 194, no. 1, pp. 365–384, Nov. 2010.

[96] S. Haspeslagh, “The First Nurse Rostering Competition 2010,” 2010. [Online]. Available: https://www.kuleuven-kulak.be/nrpcompetition.

[97] B. Mccollum *et al.*, "Setting the Research Agenda in Automated Timetabling : The Second International Time- tabling Competition," no. August 2007.

[98] G. Vanden Berghe and G. Vanden Berghe, "An advanced model and novel meta-heuristic solution methods to personnel scheduling in healthcare," 2002.

[99] E. K. Burke, T. Curtois, R. Qu, G. Vanden-berghe, and G. Vanden Berghe, "Problem model for nurse rostering benchmark instances," pp. 1–29, 2008.

[100] C. Valouxis, C. Gogos, G. Goulas, P. Alefragis, and E. Housos, "A systematic two phase approach for the nurse rostering problem," *Eur. J. Oper. Res.*, vol. 219, no. 2, pp. 425–433, 2012.

[101] R. A. M. Santos, H. G., Toffolo, T. A. M., Ribas, S., & Gomes, "Integer Programming Techniques for the Nurse Rostering Problem," in *Practice and Theory of Automated Timetabling*, 2010, pp. 256–283.

[102] IBM, CPLEX 12.2 User's Manual. 2011.

[103] "COIN-OR Linear programming solver," 2010. [Online]. Available: https://projects.coin-or.org/Clp.

[104] E. K. Burke, T. Curtois, G. Post, R. Qu, B. Veltman, and C. E. Burke, "A hybrid heuristic ordering and variable neighbourhood search for the nurse rostering problem," *Eur. J. Oper. Res.*, vol. 188, no. 2, pp. 330–341, 2008.

[105] M. Stølevik, T. E. Nordlander, A. Riise, H. Frøyseth, A. Riise, and T. E. Nordlander, "A hybrid approach for solving real-world nurse rostering problems," in *International Conference on Principles and Practice of Constraint Programming*, 2011, pp. 85–99.

[106] Z. Lü and J.-K. Hao, "Adaptive neighborhood search for nurse rostering," *Eur. J. Oper. Res.*, vol. 218, no. 3, pp. 865–876, May 2012.

[107] S. N. Vu, M. H. N. Nguyen, L. M. Duc, C. Baril, V. Gascon, and T. B. Dinh, "Iterated local search in nurse rostering problem," in *Proceedings of the Fourth Symposium on Information and Communication Technology - SoICT '13*, 2013, pp. 71–80.

[108] I. P. Solos, I. X. Tassopoulos, and G. N. Beligiannis, "A Generic Two-Phase Stochastic Variable Neighborhood Approach for Effectively Solving the Nurse Rostering Problem," *Algorithms*, vol. 6, no. 2, pp. 278–308, May 2013.

[109] B. Bilgin, P. Demeester, M. M\is\ir, W. Vancroonenburg, G. Vanden Berghe, and T. Wauters, "A hyper-heuristic combined with a greedy shuffle approach to the nurse rostering competition," in *Proceedings of the 8th International Conference on Practice and Theory of Automated Timetabling*, 2010, pp. 1–6.

[110] K. Nonobe, "INRC2010 : An Approach Using a General Constraint Optimization Solver," *First Int. Nurse Rostering Compet. (INRC 2010)*, pp. 1–2, 2010.

[111] K. Nonobe and T. Ibaraki, "A tabu search approach to the constraint satisfaction problem as a general problem solver," *Eur. J. Oper. Res.*, vol. 106, no. 2, pp. 599–623, 1998.

[112] M. Hadwan, M. Ayob, N. R. Sabar, and R. Qu, “A harmony search algorithm for nurse rostering problems,” *Inf. Sci. (Ny).*, vol. 233, no. January, pp. 126–140, 2013.

[113] P. Brucker, E. K. Burke, T. Curtois, R. Qu, G. Vanden Berghe, and G. Vanden Berghe, “A shift sequence based approach for nurse scheduling and a new benchmark dataset,” *J. Heuristics*, vol. 16, no. 4, pp. 559–573, Nov. 2008.

[114] E. Burke, P. Cowling, P. De Causmaecker, and G. Vanden Berghe, “A memetic approach to the nurse rostering problem,” *Appl. Intell.*, vol. 15, no. 3, pp. 199–214, 2001.

[115] M. a. Awadallah, A. T. Khader, M. A. Al-Betar, and A. L. Bolaji, “Nurse Scheduling Using Harmony Search,” *2011 Sixth Int. Conf. Bio-Inspired Comput. Theor. Appl.*, pp. 58–63, Sep. 2011.

[116] M. a. Awadallah, A. T. Khader, M. A. Al-Betar, and A. L. Bolaji, “Global best Harmony Search with a new pitch adjustment designed for Nurse Rostering,” *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 25, no. 2, pp. 145–162, Jul. 2013.

[117] M. G. H. H. Omran and M. Mahdavi, “Global-best harmony search,” *Appl. Math. Comput.*, vol. 198, no. 2, pp. 643–656, May 2008.

[118] A. L. Awadallah, Mohammed A and Khader, Ahamad Tajudin and Al-Betar, Mohammed Azmi and Bolaji, “Harmony search with greedy shuffle for nurse rostering,” *Int. J. Nat. Comput. Res.*, vol. 3, no. 2, pp. 22–42, 2012.

[119] E. K. Burke, T. Curtois, R. Qu, and G. Vanden Berghe, “A Time Pre-defined Variable Depth Search for Nurse Rostering 1 Introduction,” *INFORMS J. Comput.*, 2007.

[120] E. Burke *et al.*, “A hybrid tabu search algorithm for the nurse rostering problem,” *Asia-Pacific Conf. Simulated Evol. Learn.*, no. Simulated evolution and learning. Springer Berlin Heidelberg, pp. 187–194, 1999.

[121] A. L. Awadallah, Mohammed A and Khader, Ahamad Tajudin and Al-Betar, Mohammed Azmi and Bolaji, “Hybrid Harmony Search for Nurse Rostering Problems,” *Comput. Intell. Sched. (SCIS), 2013 IEEE Symp.*, pp. 60–67, 2013.

[122] P. Cowling, G. Kendall, and E. Soubeiga, “A hyperheuristic approach to scheduling a sales summit,” in *International Conference on the Practice and Theory of Automated Timetabling*, 2000, pp. 176–190.

[123] E. K. Burke *et al.*, “Hyper-heuristics: A survey of the state of the art,” *J. Oper. Res. Soc.*, vol. 64, no. 12, pp. 1695–1724, 2013.

[124] H. Fisher and G. Thompson, “Probabilistic learning combinations of local job-shop scheduling rules,” *Ind. Sched.*, vol. 3, no. 2, pp. 225–251, 1963.

[125] H. Fang and P. Ross, “A Promising Hybrid GA/Heuristic Approach for Open-Shop Scheduling Problems In Proceedings of the 11th European Conference on Arti cial Intelligence, John Wiley and Sons, 1994, pages 590{594.,” no. 699, 1994.

[126] E. Hart, P. Ross, and J. Nelson, “Solving a real-world problem using an evolving heuristically driven schedule builder.,” *Evol. Comput.*, vol. 6, no. 1, pp. 61–80, Jan. 1998.

[127] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, “A graph-based hyper-heuristic for educational timetabling problems,” *Eur. J. Oper. Res.*, vol. 176, no. 1, pp. 177–192, Jan. 2007.

[128] S. Petrovic and R. Qu, “Case-based reasoning as a heuristic selector in a hyper-heuristic for course timetabling problems,” vol. 82, pp. 336–340, 2002.

[129] E. K. Burke *et al.*, “Exploring Hyper-heuristic Methodologies with Genetic Programming,” *Comput. Intell.*, no. Computational Intelligence. Springer Berlin Heidelberg, pp. 177–201, 2009.

[130] A. Fukunaga, “Automated discovery of composite SAT variable-selection heuristics,” in *AAAI/IAAI*, 2002, no. AAAI/IAAI, pp. 641–648.

[131] A. S. Fukunaga, “Automated discovery of local search heuristics for satisfiability testing.,” *Evol. Comput.*, vol. 16, no. 1, pp. 31–61, Jan. 2008.

[132] J. H. Drake, N. Kililis, and E. Özcan, “Generation of VNS components with grammatical evolution for vehicle routing,” in *European Conference on Genetic Programming*, 2013, pp. 25–36.

[133] M. Hyde, “A genetic programming hyper-heuristic approach to automated packing,” University of Nottingham, 2010.

[134] E. K. Burke, M. R. Hyde, and G. Kendall, “Grammatical Evolution of Local Search Heuristics,” *IEEE Trans. Evol. Comput.*, vol. 16, no. 3, pp. 406–417, Jun. 2012.

[135] E. Glanville, Ranulph and Griffiths, David and Baron, Philip and Drake, John H and Hyde, Matthew and Ibrahim, Khaled and Ozcan, “A genetic programming hyper-heuristic for the multidimensional knapsack problem,” *Kybernetes*, vol. 43, no. 9/10, pp. 1500–1511, 2014.

[136] C. Y. Chan, F. Xue, W. H. Ip, and C. F. Cheung, “A Hyper-heuristic Inspired by Pearl Hunting Pearl Hunter : An Inspired Hyper-heuristic,” pp. 1–5.

[137] S. Burke, EK and Hart, E and Kendall, G and Newall, J and Ross, P and Shulenburg, “Hyper-heuristics: An emerging direction in modern search technology,” *Handb. Metaheuristics*, vol. 2, pp. 457–474, 2003.

[138] P. Cowling, G. Kendall, and E. Soubeiga, “A Parameter-Free Hyperheuristic for Scheduling a Sales Summit,” in *Proceedings of the 4th metaheuristic international conference*, 2001, vol. 1101, pp. 4–9.

[139] P. Cowling, G. Kendall, and E. Soubeiga, “Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation,” *Work. Appl. Evol. Comput.*, pp. 1–10, 2002.

[140] E. Soubeiga, “Development and application of hyperheuristics to personnel scheduling,” University of Nottingham, 2003.

[141] O. Committee, “CHeSC : Cross-Domain Heuristic Search Competition The HyFlex Framework CHeSC : Cross-Domain Heuristic Search Competition,” pp. 23–25, 2011.

[142] G. Ochoa and T. Curtois, “A HyFlex Module for the Permutation Flow Shop Problem,” pp. 1–4.

[143] T. Curtois, G. Ochoa, M. Hyde, and J. A. Vázquez-rodríguez, “A HyFlex Module for the Personnel Scheduling Problem,” pp. 1–12, 2011.

[144] T. Curtois *et al.*, “A hyflex module for the max-sat problem,” *Univ. Nottingham, Tech. Rep*, pp. 1–5, 2011.

[145] M. Hyde, G. Ochoa, V. Antonio, and T. Curtois, “A HyFlex Module for the One Dimensional Bin Packing Problem,” pp. 1–5.

[146] J. Swan, E. Özcan, and G. Kendall, “Hyperion - a recursive hyper-heuristic framework,” in *International Conference on Learning and Intelligent Optimization*, 2011, pp. 616–630.

[147] J. Swan, J. Woodward, E. Özcan, G. Kendall, and E. Burke, “Searching the Hyper-heuristic Design Space,” *Cognit. Comput.*, vol. 6, no. 1, pp. 1–13, Feb. 2013.

[148] P. Ross, “Hyper-heuristics,” *Search Methodol.*, pp. 529–556, 2005.

[149] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, “Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one,” *Proc. 9th Annu. Conf. Genet. Evol. Comput.*, vol. 14, no. 6, pp. 1559–1565, 2007.

[150] E. Özcan, B. Bilgin, E. E. Korkmaz, and K. İstanbul, “A comprehensive analysis of hyper-heuristics,” *Intell. Data Anal.*, vol. 12, no. 1, pp. 3–23, 2008.

[151] E. K. Burke *et al.*, “A Classification of Hyper-heuristic Approaches,” *Handb. metaheuristics*, no. Handbook of metaheuristics. Springer US, pp. 449–468, 2010.

[152] O. Roeva, T. Slavov, S. Fidanova, and and S. F. Roeva Olympia, Tsonyo Slavov, “Population-based vs. single point search meta-heuristics for a pid controller tuning,” *Handb. Res. Nov. soft Comput. Intell. algorithms theory Pract. Appl. IGI Glob. Pennsylvania*, vol. Handbook o, pp. 200–233, 2013.

[153] P. Ross, J. G. J. Marín-Blázquez, S. Schulenburg, and E. Hart, “Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyper-heuristics,” in *Genetic and Evolutionary Computation Conference*, 2003, pp. 1295–1306.

[154] M. Ayob and G. Kendall, “A monte carlo hyper-heuristic to optimise component placement sequencing for multi head placement machine,” in *Proceedings of the international conference on intelligent technologies, InTech*, 2003, vol. 3, no. c, pp. 132–141.

[155] R. Bai and G. Kendall, “An Investigation of Automated Planograms Using a Simulated Annealing Based Hyper-heuristics,” *Metaheuristics Prog. as real Probl. solvers*, pp. 1–7, 2003.

[156] G. Kendall and E. Soubeiga, “Choice function and random hyperheuristics School of Computer Science & IT University of Nottingham , Nottingham Peter Cowling Department of

Computing , University of Bradford Bradford BD7 1DP , UK ; Peter.Cowling@scm.brad.ac.uk," vol. Proceeding, pp. 2–6, 2002.

[157] E. K. Burke, G. Kendall, M. Misir, and E. Özcan, "Monte Carlo hyper-heuristics for examination timetabling," *Ann. Oper. Res.*, vol. 196, no. 1, pp. 73–90, Sep. 2010.

[158] M. Carter, G. Laporte, and S. Lee, "Examination timetabling: Algorithmic strategies and applications," *J. Oper. Res. Soc.*, 1996.

[159] E. Ozcan, Y. Bykov, E. Özcan, Y. Bykov, M. Birben, and E. K. Burke, "Examination timetabling using late acceptance hyper-heuristics," *2009 IEEE Congr. Evol. Comput.*, pp. 997–1004, 2009.

[160] K. Mcclymont, H. Building, E. Ex, and E. C. Keedwell, "Markov Chain Hyper-heuristic (MCHH): an Online Selective Hyper-heuristic for Multi-objective Continuous Problems," pp. 2003–2010, 2011.

[161] E. Burke, G. Kendall, D. L. L. Silva, R. O'Brien, and E. Soubeiga, "An Ant Algorithm Hyperheuristic for the Project Presentation Scheduling Problem," *2005 IEEE Congr. Evol. Comput.*, vol. 3, pp. 2263–2270, 2005.

[162] P.-C. Chen, G. Kendall, and G. Vanden Berghe, "An Ant Based Hyper-heuristic for the Travelling Tournament Problem," *2007 IEEE Symp. Comput. Intell. Sched.*, pp. 19–26, Apr. 2007.

[163] E. K. E. K. K. Burke, G. Kendall, and E. Soubeiga, "A Tabu-Search Hyperheuristic for Timetabling and Rostering," *J. Heuristics*, vol. 9, no. 6, pp. 451–470, Dec. 2003.

[164] G. Kendall and N. M. Hussin, "Tabu Search Hyper-heuristic Approach to the Examination Timetabling Problem at University Technology MARA," in *International Conference on the Practice and Theory of Automated Timetabling*, 2004, no. 1996, pp. 270–293.

[165] G. Kendall and N. M. Hussin, "An investigation of a tabu search based hyper-heuristic for examination timetabling hyper-heuristics," *Multidiscip. Sched. Theory Appl.*, no. November, pp. 309–328, 2005.

[166] W. Crowston, F. Glover, G. Thompson, and J. Trawick, "Probabilistic and Parametric Learning Methods for the Job Shop Scheduling Problem," *GSIA*, 1964.

[167] H. Fang, P. Ross, and D. Corne, "A Promising genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems Appears in: Proceedings of the Fifth International Conference on Genetic Algorithms, S. Forrest (ed.), San Mateo: Morgan Kaufmann, 1993, pages 3," no. 623, 1993.

[168] E. López-camacho, H. Terashima-marín, and P. Ross, "A hyper-heuristic for solving one and two-dimensional bin packing problems," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, 2011, pp. 257–258.

[169] P. Ross *et al.*, "Hyper-heuristics: learning to combine simple heuristics in bin-packing problems.," in *GECCO*, 2002, pp. 942–948.

[170] N. Pillay, “Evolving hyper-heuristics for the uncapacitated examination timetabling problem,” *J. Oper. Res. Soc.*, vol. 63, no. 1, pp. 47–58, Apr. 2011.

[171] E. K. Burke, J. P. Newall, and R. F. Weare, “A Simple Heuristically Guided Search for the Timetable Problem A Heuristically Guided Random Search.”

[172] R. Els and N. Pillay, “An evolutionary algorithm hyper-heuristic for producing feasible timetables for the curriculum based university course timetabling problem,” *2010 Second World Congr. Nat. Biol. Inspired Comput.*, pp. 460–466, Dec. 2010.

[173] K. Socha, J. Knowles, and M. Sampels, “A max-min ant system for the university course timetabling problem,” in *Ant algorithms*, 2002, pp. 1–13.

[174] R. Qu and E. K. Burke, “Hybridizations within a graph-based hyper-heuristic framework for university timetabling problems,” *J. Oper. Res. Soc.*, vol. 60, no. 9, pp. 1273–1285, 2009.

[175] N. R. Sabar, M. Ayob, R. Qu, and G. Kendall, “A graph coloring constructive hyper-heuristic for examination timetabling problems,” *Appl. Intell.*, vol. 37, no. 1, pp. 1–11, Aug. 2011.

[176] N. Pillay and W. Banzhaf, “A study of heuristic combinations for hyper-heuristic systems for the uncapacitated examination timetabling problem,” *Eur. J. Oper. Res.*, vol. 197, no. 2, pp. 482–491, Sep. 2009.

[177] M. Oltean and D. Dumitrescu, “Evolving TSP heuristics using multi expression programming,” *Comput. Sci. 2004*, no. 0, pp. 670–673, 2004.

[178] M. Oltean, “Multi Expression Programming,” pp. 1–28, 2006.

[179] G. Reinelt, “TSPLIB—A traveling salesman problem library,” *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

[180] R. Poli, J. Woodward, and E. K. Burke, “A histogram-matching approach to the evolution of bin-packing strategies,” *2007 IEEE Congr. Evol. Comput.*, pp. 3500–3507, Sep. 2007.

[181] C. Dimopoulos and a. M. S. A. M. S. Zalzala, “Investigating the use of genetic programming for a classic one-machine scheduling problem,” *Adv. Eng. Softw.*, vol. 32, no. 6, pp. 489–498, Jun. 2001.

[182] E. Montagne, “Sequencing with time delay costs,” *Ind. Eng. Res. Bull. Arizona State Univ.*, vol. 5, pp. 20–31, 1969.

[183] C. D. Geiger, R. Uzsoy, H. Aytu\ug, and H. Aytuğ, “Rapid Modeling and Discovery of Priority Dispatching Rules: An Autonomous Learning Approach,” *J. Sched.*, vol. 9, no. 1, pp. 7–34, Feb. 2006.

[184] D. Jakobovi, D. Jakobović, and L. Budin, “Dynamic scheduling with genetic programming,” in *European Conference on Genetic Programming*, 2006, pp. 73–84.

[185] N. B. Ho and J. C. Tay, “Evolving Dispatching Rules for solving the Flexible Job-Shop Problem,” *2005 IEEE Congr. Evol. Comput.*, vol. 3, pp. 2848–2855, 2005.

[186] J. C. Tay and N. B. Ho, “Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems,” *Comput. Ind. Eng.*, vol. 54, no. 3, pp. 453–473, Apr. 2008.

[187] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward, “The scalability of evolved on line bin packing heuristics,” *2007 IEEE Congr. Evol. Comput.*, pp. 2530–2537, Sep. 2007.

[188] E. K. Burke, M. Hyde, G. Kendall, and J. Woodward, “A Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics,” pp. 1–17.

[189] S. Allen, E. K. Burke, M. Hyde, and G. Kendall, “Evolving reusable 3d packing heuristics with genetic programming,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 931–938.

[190] S. Allen, E. Burke, and G. Kendall, “A new hybrid placement strategy for the three-dimensional strip packing problem,” 2009.

[191] J. Xie, Y. Mei, A. T. Ernst, X. Li, and A. Song, “A genetic programming-based hyper-heuristic approach for storage location assignment problem,” in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 3000–3007.

[192] E. Burke, M. Dror, S. Petrovic, and R. Qu, “Hybrid graph heuristics within a hyper-heuristic approach to exam timetabling problems,” in *The next wave in computing, optimization, and decision technologies*, Springer, 2005, pp. 79–91.

[193] E. Burke, A. Eckersley, and B. Mccollum, “Computer Science Technical Report No . NOTTCS-TR-2006-2 Hybrid Variable Neighbourhood Approaches to University Exam Timetabling Hybrid Variable Neighbourhood Approaches to University Exam Timetabling,” 2006.

[194] H. Asmuni, E. K. Burke, J. M. Garibaldi, and B. McCollum, “Fuzzy Multiple Heuristic Ordering for Examination Timetabling,” *Int. Conf. Pract. Theory Autom. Timetabling*, pp. 334–353, 2004.

[195] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, “SATenstein: Automatically Building Local Search SAT Solvers from Components.,” in *IJCAI*, 2009, vol. 9, no. October, pp. 517–524.

[196] S. Nguyen, M. Zhang, and M. Johnston, “A genetic programming based hyper-heuristic approach for combinatorial optimisation,” in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 2011, pp. 1299–1306.

[197] R. E. Keller and R. Poli, “Cost-Benefit Investigation of a Genetic-Programming Hyperheuristic A Linear-GP Hyperheuristic,” pp. 13–24, 2008.

[198] R. E. Keller and A. G. Hyperheuristic, “Linear Genetic Programming of Parsimonious Metaheuristics.”

[199] R. E. Keller and R. Poli, “Improved Benchmark Results from Subheuristic Search,” 2008.

[200] R. E. Keller, R. Poli, A. G. Hyperheuristic, and R. Keller, Robert E and Poli, “Linear genetic programming of parsimonious metaheuristics,” *2007 IEEE Congr. Evol. Comput.*, pp. 4508–4515, 2007.

[201] R. E. Keller and R. Poli, “Self-adaptive hyperheuristic and greedy search,” *2008 IEEE Congr. Evol. Comput. (IEEE World Congr. Comput. Intell.*, pp. 3801–3808, Jun. 2008.

[202] M. Bader-el-den and R. Poli, “Generating SAT local-search heuristics using a GP hyper-heuristic framework,” in *International Conference on Artificial Evolution (Evolution Artificielle)*, 2007, pp. 37–49.

[203] J. Gottlieb, E. Marchiori, and C. Rossi, “Evolutionary algorithms for the satisfiability problem,” *Evol. Comput.*, vol. 10, no. 1, pp. 35–50, 2002.

[204] N. R. Sabar, M. Ayob, G. Kendall, R. Qu, S. Member, and R. Qu, “Grammatical evolution hyper-heuristic for combinatorial optimization problems,” *IEEE Trans. Evol. Comput.*, vol. 17, no. 6, pp. 840–861, 2013.

[205] P. Cowling, G. Kendall, and E. Soubeiga, “Hyperheuristics: A robust optimisation method applied to nurse scheduling,” in *International Conference on Parallel Problem Solving from Nature*, 2002, pp. 851–860.

[206] K. A. Dowsland, “Nurse Scheduling with tabu search and strategic oscillation,” *Eur. J. Oper. Res.*, vol. 106, no. 2, pp. 393–407, 1998.

[207] K. A. Dowsland and W. Lane, “An Indirect Genetic Algorithm for a Nurse Scheduling Problem,” vol. 31, no. 5, pp. 761–778, 2004.

[208] E. Burke and E. Soubeiga, “Scheduling nurses using a tabu-search hyperheuristic,” *Proc. 1st Multidiscip. Int. Conf. Sched. Theory Appl. (MISTA 2003), Nottingham, UK*, no. Proceedings of the 1st multidisciplinary international conference on scheduling: Theory and applications (MISTA 2003), Nottingham, UK. 2003, pp. 1–22, 2003.

[209] R. Bai, E. K. Burke, G. Kendall, J. Li, and B. McCollum, “A Hybrid Evolutionary Approach to the Nurse Rostering Problem,” *IEEE Trans. Evol. Comput.*, vol. 14, no. 4, pp. 580–590, Aug. 2010.

[210] B. Bilgin, P. De Causmaecker, P. De Causmaecker, and G. Vanden Berghe, “A Hyperheuristic Approach to Belgian Nurse Rostering Problems,” *Proc. 4th Multidiscip. Int. Conf. Sched. Theory Appl.*, no. Proceedings of the 4th Multidisciplinary International Conference on Scheduling: Theory and Applications. 2009, pp. 10–12, 2009.

[211] B. Bilgin *et al.*, “Local search neighbourhoods for dealing with a novel nurse rostering model,” *Ann. Oper. Res.*, vol. 194, no. 1, pp. 33–57, Nov. 2012.

[212] G. Vanden Bilgin, Burak and De Causmaecker, Patrick and Rossie, Benot and Berghe, “Local Search Neighbourhoods to Deal with a Novel Nurse Rostering Model,” *Ann. Oper. Res.*, vol. 194, no. 1, pp. 33–57, 2012.

[213] B. Bilgin, P. Demeester, M. Misir, W. Vancroonenburg, G. Vanden Berghe, and G. Vanden, “One hyper-heuristic approach to two timetabling problems in health care,” *J. Heuristics*, vol. 18, no. 3, pp. 401–434, 2012.

[214] K. Anwar, M. A. Awadallah, A. T. Khader, and M. A. Al-Betar, “Hyper-heuristic approach for solving nurse rostering problem,” in *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*, 2014, pp. 1–6.

[215] O. Committee, “CHeSC : Cross-Domain Heuristic Search Competition ASAP Default Hyper-heuristics CHeSC : Cross-Domain Heuristic Search Competition,” pp. 23–25, 2011.

[216] M. Mısır *et al.*, “An intelligent hyper-heuristic framework for chesc 2011,” *Learn. Intell. Optim.*, pp. 461–466, 2012.

[217] P.-C. Hsiao, T.-C. Chiang, and L.-C. Fu, “A VNS-based hyper-heuristic with adaptive computational budget of local search,” *2012 IEEE Congr. Evol. Comput.*, pp. 1–8, Jun. 2012.

[218] M. Larose, “A hyper-heuristic for the chesc 2011,” in *The 53rd Annual Conference of the UK Operational Research Society (OR53)*, 2011, pp. 1–2.

[219] D. Meignan, A. Koukam, and J. Créput, “Coalition-based metaheuristic: a self-adaptive metaheuristic using reinforcement learning and mimetism,” *J. Heuristics*, 2010.

[220] A. Lehrbaum, “A New Hyperheuristic Algorithm for Cross-Domain Search Problems,” vol. 2011, 2011.

[221] R. Poli and M. Graff, “There is a free lunch for hyper-heuristics, genetic programming and computer scientists,” in *European Conference on Genetic Programming*, 2009, pp. 195–207.

[222] M. O’Neil et al., Grammatical evolution: evolutionary automatic programming in an arbitrary language. Norwell, MA: Kluwer Academic Publishers, 2003.

[223] S. Luke and L. Panait, “A survey and comparison of tree generation algorithms,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001, pp. 81–88.

[224] C. Johnson, “Basic Research Skills in Computing Science.” [Online]. Available: http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/basics.html .

[225] A. A. Constantino, D. Landa-Silva, E. L. de Melo, C. F. X. de Mendonça, D. B. Rizzato, and W. Romão, “A heuristic algorithm based on multi-assignment procedures for nurse scheduling,” *Ann. Oper. Res.*, vol. 218, no. 1, pp. 165–183, Apr. 2014.

[226] E. Burke and Y. Bykov, “A late acceptance strategy in hill-climbing for exam timetabling problems,” in *PATAT 2008 Conference, Montreal, Canada. 2008.*, 2008, p. 7.

[227] Y. Bykov and S. Petrovic, “An initial study of a novel Step Counting Hill Climbing heuristic applied to timetabling problems,” in *Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA-13)*, 2013, pp. 691–693.

[228] M. Misir, “A selection hyper-heuristic for scheduling deliveries of ready-mixed concrete,” *Proc. Metaheuristics Int. Conf. (MIC 2011)*, pp. 289–298, 2011.

[229] E. K. Burke and Y. Bykov, “The late acceptance hill-climbing heuristic,” *Univ. Stirling, Tech. Rep*, no. June, p. 19, 2012.

[230] S. Bykov, Yuri and Petrovic, “A Step Counting Hill Climbing Algorithm,” *Nottingham Univ. Bus. Sch. Res. Pap. Ser.*, vol. 10, p. 23, 2013.

[231] F. Hutter, H. H. H. Hoos, K. Leyton-Brown, and T. Stützle, “ParamILS: an automatic algorithm configuration framework,” *J. Artif. Intell. Res.*, vol. 36, no. 1, pp. 267–306, 2009.

[232] R. N. Kackar, “Off-line quality control, parameter design, and the Taguchi method,” in *Quality Control, Robust Design, and the Taguchi Method*, Springer, 1989, pp. 51–76.

[233] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, “Automated algorithm tuning using F-Races: Recent developments,” in *Proceedings of MIC*, 2009, vol. 9, pp. 1–10.

Appendix A

This appendix describes how to run the two programs.

A.1 Program requirements

Java 1.6 must be installed in order to use the program (http://java.com/en/download). Once Java is installed on the machine the two jar files should be executable. Copy the folder "Executables" from the CD to your computer. These are called SPHH.jar and GPHH.jar. For linux machines, navigate to the folder, right click on the jar file and click "Open in Terminal". Then type the following command: java –jar filename.jar. For windows users please use the two included ".bat" files are provided these are called SPHH.bat and GPHH.bat. For linux users two ".sh" files are provided these are called SPHH.sh and GPHH.sh.

A.2 SPHH

"SPHH.jar" is the selections perturbative hyper-heuristic presented in **Chapter 8.** This program can be seen in Figure A.1. SPHH includes a number of preset options for population size, genetic operator rates, tournament size, generation limit, convergence limit and the initial individual length values. The tournament size cannot be set higher than the population size. An option is included to output to file (each file name is unique), this will create a text file in the subfolder called "Results". If the checkbox is not checked no file will be written and the data will be displayed through the console. A further option is available to run SPHH before multithreading was added. Finally the instance to run the SPHH algorithm on can be selected. A label will display the word "Running" while a run is ongoing and a pop up message will confirm the run has finished.

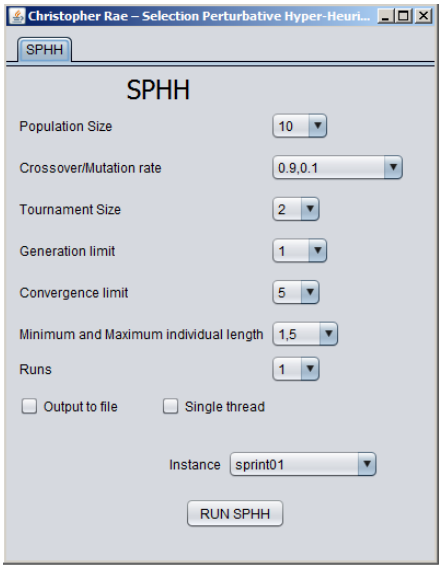

Figure A.1 SPHH program

A.3 GPHH

"GPHH.jar" is the generative perturbative hyper-heuristic presented in **Chapter 9**. GPHH includes two tabs, the first seen in is the interface to evolve a new heuristic and the second seen in is the interface to run an evolved heuristic. Evolved heuristics are available in the subfolder "EvolvedHeuristics".

The first tab as seen in Figure A.2, “GPHH” is used to evolve a heuristic. This tab includes a number of preset options for population size, genetic operator rates, tournament size, generation limit, offspring for each generation, the IF-C value, probability swap value and the evaluation time for Algorithm 9.2 (the value is set to milliseconds such that 1 = 100ms). The tournament size and offspring for each generation must not exceed the population size. An option is included to output to file (each file name is unique), this will create a text file in the subfolder called “Results”. A further option is available to run GPHH using 3 instances, this will display two additional combo boxes shown in Figure A.4. The instances do not have to be unique but using this option will triple the runtime of the program. When a run finishes the evolved heuristic will be written to the folder “EvolvedHeuristics”. This new heuristic can be run using the second program tab. A pop up message will also confirm the run has finished.

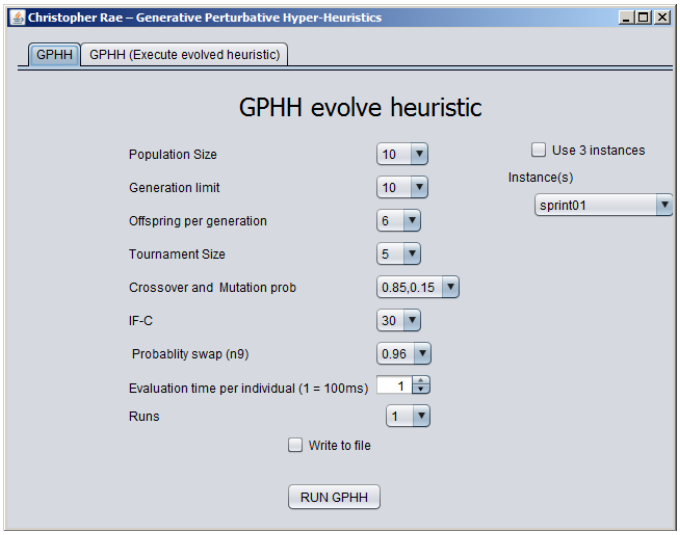

Figure A.2 GPHH tab 1 evolve new heuristic

The second tab as seen in Figure A.3, “GPHH (Execute evolved heuristic) is used to run an evolved heuristic. This tab also presents a number of preset options for the probability swap value, the time period to apply the evolved heuristic to the chosen instance and the IF-C value. The competition benchmarking tool is included in the folder “Benchmarking”. This tool will give a set of suggested time values for solving each track of instance. For this program the default values of 8 seconds for sprint instances, 8 minutes for medium instances and 1 hour for long instances are set. The time values can be changed by the user and will only update to default values when selecting a new instance or inputting a time value of 0. The heuristic file you wish to use can be selected and the instance you wish to apply the evolved heuristic to can be chosen. An option is included to output to file (each file name is unique), this will create a text file in the subfolder called “Results”.

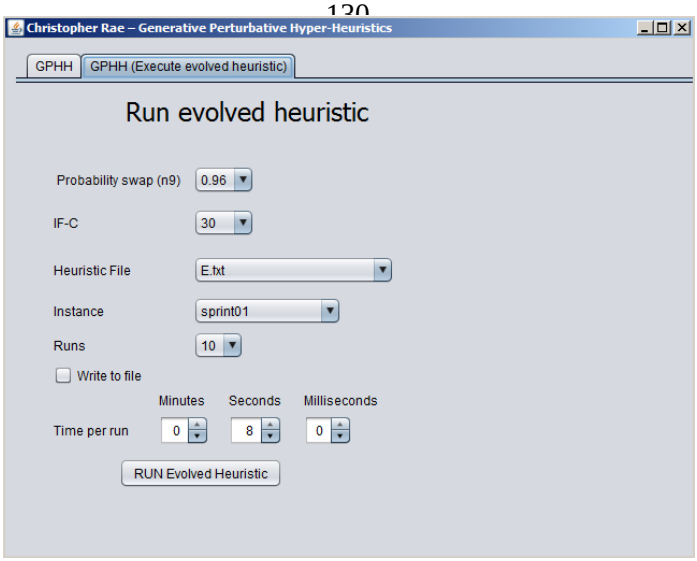

Figure A.3 GPHH tab 2 run evolved heuristic

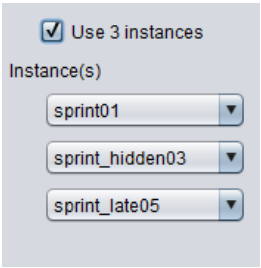

Figure A.4 Using 3 instances option

A.4 Running an experiment

First the parameters must be set. Default values have been provided in order to quickly demonstrate the two approaches. The parameters can be changed and the parameters used for the simulations are available to the user. SPHH and GPHH are multithreaded and will attempt to use the maximum number of threads available, for example a computer with 4 cores and no hyper-threading will have 4 threads but a computer with 4 cores and hyper-threading will have 8 threads available. Both approaches will try to use the maximum number of available threads. The exception is the single threaded version of SPHH. An error message will be displayed if a parameter is not set properly for example, in Figure A.6 the evaluation time for running an evolved heuristic was set to 0. It is recommended that you run a single instance of a program at a time. While a program is running a label will display the word “Running” this can be seen in Figure A.5. When this label no longer displays “Running” the run has finished.

Finally an instance must be selected from the INRC2010 benchmark data set and a number of runs must be chosen.

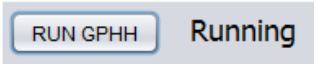

Figure A.5 Displaying "Running" label while executing program

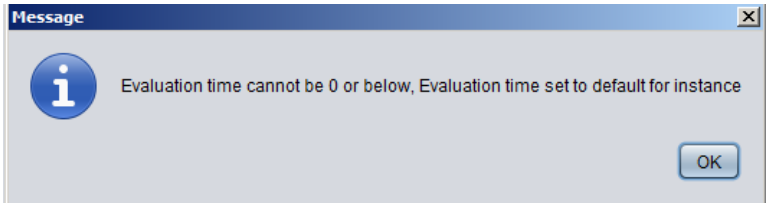

Figure A.6 Example of an error message pop up

Appendix B

This appendix contains tables of results pertaining to Chapter 10 section 10.2.

B.1 GPHH related results tables

Table B.1 Minimum values for sprint instances

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
sprint_early1	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early2	58	58	58	58	58	58	58	58	58	58	58	58	58	58
sprint_early3	46	51	51	51	51	51	51	51	51	51	51	51	51	51
sprint_early4	59	59	59	59	59	59	59	59	59	59	59	59	59	60
sprint_early5	58	58	58	58	58	58	58	58	58	58	58	58	58	58
sprint_early6	54	54	54	54	54	54	54	54	54	54	54	54	54	54
sprint_early7	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early8	56	56	56	56	56	56	56	56	56	56	56	56	56	56
sprint_early9	55	55	55	55	55	55	55	55	55	55	55	55	55	56
sprint_early10	52	52	52	52	52	52	52	52	52	52	52	52	52	52
sprint_late1	37	38	40	38	39	39	39	39	39	38	38	39	39	40
sprint_late2	42	43	43	43	43	42	42	43	43	43	43	43	43	45
sprint_late3	48	49	49	48	49	49	49	50	49	49	49	49	50	50
sprint_late4	73	80	78	79	76	80	80	75	77	78	77	77	75	90
sprint_late5	44	45	45	45	45	45	44	45	45	45	45	45	45	47
sprint_late6	42	42	42	42	42	42	42	42	42	42	43	42	42	43
sprint_late7	42	46	47	44	47	44	45	45	44	43	45	46	44	49
sprint_late8	17	17	17	17	17	17	17	17	17	17	17	17	17	17
sprint_late9	17	17	17	17	17	17	17	17	17	17	17	17	17	17
sprint_late10	43	45	48	45	47	46	47	47	46	46	46	48	47	50
sprint_hidden1	32	33	33	33	33	32	33	33	33	33	34	34	32	37
sprint_hidden2	32	32	32	32	32	32	32	33	32	32	32	32	32	35
sprint_hidden3	62	63	64	62	63	63	63	63	62	62	63	62	64	64
sprint_hidden4	66	66	67	66	66	67	67	68	66	66	67	68	67	68
sprint_hidden5	59	60	59	60	59	59	61	60	59	60	60	60	59	60
sprint_hidden6	130	144	172	150	160	166	178	154	163	149	170	166	167	160
sprint_hidden7	153	161	162	156	161	156	157	163	166	161	161	158	158	171
sprint_hidden8	204	211	221	217	215	211	216	213	215	214	220	221	215	239
sprint_hidden9	338	345	348	343	346	340	343	345	339	352	347	349	340	361
sprint_hidden10	306	306	306	306	306	306	312	311	306	306	306	318	306	324

Table B.2 Average values for sprint instances

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
sprint_early1	56.57	56.5	56.53	56.63	56.53	57.03	56.9	56.3	56.37	56.8	56.8	56.87	57.43
sprint_early2	58.67	58.6	58.53	58.7	58.23	59.23	59	58.2	58.3	58.8	58.77	59.17	59.93
sprint_early3	52.17	51.93	52	52.37	51.47	52.5	52.43	51.87	51.97	52.3	52.47	52.5	53.2
sprint_early4	60.33	60.43	60.23	60.63	60.07	60.73	60.97	60.1	60.3	60.7	60.33	61.03	62.23
sprint_early5	58.03	58.03	58	58.1	58.03	58.17	58.07	58.03	58.03	58.03	58	58.17	58.7
sprint_early6	54.03	54.13	54.17	54.27	54.03	54.8	54.77	54.1	54.1	54.43	54.33	54.6	55.3
sprint_early7	56.77	57.03	56.8	56.87	56.5	57.63	57.4	56.6	56.83	57.1	57	57.8	57.87
sprint_early8	56.87	56.37	56.47	56.67	56.23	56.8	56.73	56.37	56.53	56.6	56.53	57	56.93
sprint_early9	55.97	56.37	56.1	56.3	56	56.47	56.43	56.1	56	56.17	56.87	56.83	58.13
sprint_early10	52.9	53.33	52.8	53	52.77	53.3	52.93	52.87	52.6	53	53.17	53.67	54.53
sprint_late1	40.63	41.9	40.87	41.07	40.73	41.37	41.47	40.87	40.63	41.43	41.8	42.1	44.37
sprint_late2	45.03	45.53	45.03	45.27	45.2	45.4	45.47	45.2	45.53	45.4	45.57	46.07	49.07
sprint_late3	51.4	51.67	51.37	51.53	50.73	52.3	51.6	51.57	51.13	51.27	51.83	52	55.03
sprint_late4	87.97	89.13	87.17	86.17	87.67	88.77	87.7	85.47	88.8	86.17	87.47	89	104.47
sprint_late5	46.9	47	46.5	46.97	46.73	46.97	46.73	46.87	46.47	47.03	46.97	47.37	49.37

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
sprint_late6	43.4	43.63	43.03	43.53	43.37	43.6	43.47	43.33	43.13	43.5	43.7	43.73	45.53
sprint_late7	51.63	53.63	51.57	52.2	52.03	51.07	51.17	51.47	50.07	52.33	52.13	52	59.07
sprint_late8	22.8	21.4	20.7	23.03	22	21.73	22.63	21.37	21.13	21.67	22	21.33	26.47
sprint_late9	21.87	23.23	24	21.3	22.43	21.87	22.37	21.97	22.9	23.13	23.73	23.8	27.7
sprint_late10	52.17	53.03	52.47	53.67	52.27	54.33	54.5	53	51.63	53.13	54.27	55	59.23
sprint_hidden1	35.9	37.33	36.4	37.23	35.67	37.07	36.9	36	36.23	36.37	36.43	37.13	41.3
sprint_hidden2	35.1	35.4	34.67	35	34.8	35.53	34.77	34.07	34.47	34.97	34.77	34.97	39.8
sprint_hidden3	66.67	68.17	66.97	68	67.57	67.8	67.63	66.9	67	68	67.4	68.73	72.83
sprint_hidden4	69.27	69.67	69.87	70.17	69.23	69.47	70.03	69.43	69.4	70.1	70.53	70.7	72.6
sprint_hidden5	62.97	63.5	63.03	63.83	62.93	63.7	62.63	62.93	62.77	63.67	63.23	64.4	67.8
sprint_hidden6	161.03	168.8	166.17	167.5	167.07	170.53	164.43	166.57	168.83	170.17	168	171.5	189.07
sprint_hidden7	182.37	191.77	180.37	187.37	183	187.23	182.63	185.63	184	187.43	191.07	190.67	216.4
sprint_hidden8	234.07	240.1	237.13	242.2	238.73	241.2	235.23	240.37	239.5	239.7	243	241.2	272.83
sprint_hidden9	362.9	369.2	363.87	366.83	361.17	365.57	364.37	364.6	365.4	366.53	367.1	368.1	388.9
sprint_hidden10	330.57	340.97	328.53	329.73	337.93	335.77	331.87	331.2	332.7	332.6	338.13	335.47	375.67

Table B.3 Minimum values for medium instances

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
medium_early1	240	241	241	240	241	242	243	243	241	241	242	242	243	242
medium_early2	240	241	241	241	242	241	241	241	241	241	241	241	242	241
medium_early3	236	238	238	237	238	237	238	237	238	237	238	238	238	238
medium_early4	237	239	238	238	238	238	239	239	238	238	239	238	239	239
medium_early5	303	304	304	304	304	303	305	304	304	304	304	304	305	306
medium_late1	157	176	182	182	177	180	179	175	182	177	178	176	185	201
medium_late2	18	24	28	24	27	26	29	24	22	25	25	27	28	30
medium_late3	29	35	33	32	33	34	36	34	32	32	36	33	36	39
medium_late4	35	39	39	38	39	38	38	38	37	39	38	39	38	45
medium_late5	107	139	138	130	137	134	140	136	142	139	142	135	141	172
medium_hidden1	121	148	144	148	150	148	156	138	142	146	155	149	136	174
medium_hidden2	221	258	274	263	269	254	278	262	267	271	275	270	257	302
medium_hidden3	35	41	40	37	39	38	39	38	39	39	39	37	40	49
medium_hidden4	78	88	86	86	86	87	87	87	85	87	85	84	88	91
medium_hidden5	119	140	138	134	137	139	143	137	137	130	141	143	141	146

Table B.4 Average values for medium instances

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
medium_early1	243.17	242.87	242.93	243.03	243.23	246.77	244.23	242.77	242.6	243.53	245.37	244.6	243.9
medium_early2	244.37	242.47	242.43	242.67	242.27	245.73	242.8	243.5	242.17	242.83	244.87	243.97	243.83
medium_early3	241.63	238.93	238.9	238.8	239.23	240.37	239.8	239.37	239.03	239.5	241.6	240.3	240.17
medium_early4	242	239.93	239.67	239.77	239.43	240.83	240.6	240.7	239.5	240.5	241.9	241.2	241
medium_early5	308.23	305.73	305.3	305.97	305.37	309.3	306.27	305.53	305.4	306.6	308.43	307	308.93
medium_late1	191.7	200.8	193.4	198.73	193.33	196.67	195.63	192.73	190.33	194.77	192.5	197.73	218.23
medium_late2	31.63	34.77	30.4	31.17	32.27	31.47	30.7	30.53	30.37	32.87	33.1	34.9	40.3
medium_late3	39	39.8	38.13	38.33	39.2	41.33	39.17	41.37	39.03	40.6	39.63	43.37	47.23
medium_late4	42.6	43.3	41.53	43.27	42.5	43.13	41.93	46.4	41.77	43.2	43.03	44.63	51.33
medium_late5	154.07	162.67	152.1	156.7	155	161.87	156.27	173.33	157.23	156.83	161.1	163	195.03
medium_hidden1	166.77	173.17	170.63	173.43	167.9	177.63	169.67	173.9	169.03	172.33	177.6	180.43	209
medium_hidden2	293.87	302.33	288.67	297.2	292.63	300.53	293.2	306.53	293.3	300.4	296.77	297.17	340.63
medium_hidden3	44.37	44.8	43.63	44.63	43	44.2	44.07	43.47	43.97	44.83	49.77	45.2	53.57
medium_hidden4	93.63	95.23	92.07	94.23	93.4	93.83	94.5	93.63	94.3	93.6	98.67	95.57	103.2
medium_hidden5	155.03	165.8	153.97	158.2	154.2	158.9	153.37	151.93	159.5	156.03	174.87	159.97	187.57

Table B.5 Minimum values for long instances

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
long_early1	197	197	197	197	197	197	197	197	197	197	197	197	197	197
long_early2	219	221	221	222	221	221	222	222	221	221	221	221	222	223
long_early3	240	240	240	240	240	240	240	240	240	240	240	240	240	240

Instances	BKRs	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
long_early4	303	303	303	303	303	303	303	303	303	303	303	303	303	303
long_early5	284	284	284	284	284	284	284	284	284	284	284	284	284	284
long_late1	235	254	262	257	257	255	262	260	257	250	257	259	268	283
long_late2	229	249	273	259	266	263	262	253	258	260	259	263	262	305
long_late3	220	257	268	258	263	258	255	256	262	263	266	260	256	294
long_late4	221	256	271	259	267	264	257	258	259	255	264	268	269	299
long_late5	83	99	104	97	96	96	99	93	95	98	100	98	97	123
long_hidden1	346	368	380	374	374	368	369	370	381	374	377	377	374	423
long_hidden2	89	90	91	89	91	90	90	90	91	90	90	91	92	101
long_hidden3	38	43	45	41	45	43	42	42	40	40	42	44	42	55
long_hidden4	22	23	27	25	24	27	26	27	26	25	27	25	28	37
long_hidden5	41	48	52	48	48	48	45	51	45	46	48	48	51	67

Table B.6 Average values for long instances

Instances	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
long_early1	197.13	197.07	197.2	197.47	197.03	198.4	198.2	197	197.2	198.13	197.87	198.57	197.57
long_early2	230.33	223	223.5	223.47	222.57	225.33	224.53	227.27	222.97	224.7	224.53	225.8	226.53
long_early3	240	240	240	240	240	240	240	240.67	240	240	241.37	240	240
long_early4	307.87	303.07	303	303	303.07	303.93	303.8	303.07	303.03	303.67	306.7	304.37	304.4
long_early5	289.27	284.3	284	284.03	284	284.83	284.43	284	284	284.57	284.27	285.07	285.17
long_late1	267.63	285.47	271.57	278.3	274	276.07	273.2	271.67	271.5	276.17	282.07	282.43	315.87
long_late2	273.03	296.27	275.63	285.67	277.43	278.4	276.57	273.1	277.63	282.53	284.33	289.07	326.77
long_late3	275.77	285.3	275.97	279.33	277.3	278.03	277.63	276.2	276.4	279.07	277.8	284.1	323.2
long_late4	273.47	286.47	276.4	283.33	280.3	281.97	275.77	275.43	275.4	277.87	285	291.47	327.6
long_late5	110.7	120.07	111.47	115.47	113.57	116.03	110.9	111.5	110.93	115.7	118.43	118.07	152.97
long_hidden1	396.27	407.57	397.97	401.57	396.43	399.17	399.73	445.53	397.03	399.57	403.63	414.6	466.3
long_hidden2	93.93	98.37	93.2	95.3	94.03	95.4	93.93	105.03	93.97	96.5	95.9	98	111.57
long_hidden3	48	51.8	48.27	50.57	47.8	49.6	47.77	48.47	48.17	50.27	51.37	50.4	63.07
long_hidden4	31.9	34.7	30.87	33.7	31.43	31.77	31.4	30.8	30.47	31.3	43.77	35.37	49.3
long_hidden5	56.27	63.3	55.67	57.1	55.7	58.3	56.2	54.43	56.57	58.4	58.87	59.87	79.83

Table B.7 Standard deviations GPHH

Instance	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
sprint_early1	0.57	0.57	0.73	0.76	0.63	0.76	0.84	0.47	0.61	0.85	0.85	0.78	1.07
sprint_early2	0.71	0.62	0.82	0.79	0.50	0.86	0.74	0.41	0.47	0.81	0.63	0.83	1.39
sprint_early3	0.79	0.87	0.79	0.89	0.63	0.78	0.77	0.82	0.85	0.88	0.94	0.94	1.27
sprint_early4	0.96	1.04	0.73	0.93	0.74	1.08	0.93	1.03	0.79	0.79	0.66	1.27	1.45
sprint_early5	0.18	0.18	0.00	0.31	0.18	0.38	0.25	0.18	0.18	0.18	0.00	0.46	0.79
sprint_early6	0.18	0.35	0.38	0.52	0.18	0.76	0.73	0.31	0.31	0.63	0.55	0.62	0.92
sprint_early7	0.82	0.85	0.85	0.82	0.82	0.81	0.97	0.77	0.83	0.80	0.79	1.10	0.94
sprint_early8	0.68	0.49	0.57	0.71	0.43	0.66	0.52	0.49	0.63	0.62	0.68	0.79	0.78
sprint_early9	1.03	1.07	0.92	1.06	1.08	1.14	1.19	0.99	0.95	1.02	0.97	1.26	1.28
sprint_early10	0.76	0.76	0.71	0.59	0.73	0.84	0.74	0.73	0.62	0.79	0.87	1.03	1.14
sprint_late1	1.16	1.32	1.43	1.39	1.17	1.33	1.53	1.04	1.40	1.38	1.65	1.75	2.01
sprint_late2	1.25	1.43	1.10	1.46	1.49	1.65	1.38	1.40	1.81	1.25	1.57	1.53	2.49
sprint_late3	1.22	1.60	1.19	1.53	1.14	1.66	1.04	1.19	1.38	1.26	1.84	1.36	2.59
sprint_late4	4.86	6.22	4.70	6.13	5.11	6.21	7.36	3.79	6.21	5.38	5.71	6.30	8.27
sprint_late5	0.96	1.31	1.04	1.19	1.20	1.07	1.17	1.17	1.01	1.40	1.03	1.13	1.59
sprint_late6	0.77	0.85	0.85	0.86	1.07	1.13	1.01	0.88	0.82	0.63	0.95	1.01	1.59
sprint_late7	4.09	5.83	4.30	3.90	3.86	3.91	3.42	3.50	3.53	4.02	4.10	3.54	5.71
sprint_late8	3.72	4.34	3.74	4.68	4.36	3.79	4.58	4.23	4.82	4.83	4.62	3.49	5.15
sprint_late9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sprint_late10	4.57	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sprint_hidden1	1.84	1.88	1.96	2.39	1.73	1.86	2.22	1.62	1.68	1.65	1.72	2.26	2.64
sprint_hidden2	2.07	2.01	1.81	1.62	1.71	2.30	1.89	1.44	1.68	1.69	1.99	1.96	3.23
sprint_hidden3	2.11	2.34	2.67	2.27	2.03	2.37	2.74	2.94	2.59	2.86	3.06	2.41	3.52
sprint_hidden4	1.68	1.99	1.63	1.93	1.50	1.48	1.75	1.70	1.71	1.75	1.83	1.82	2.09
sprint_hidden5	2.22	2.18	1.87	2.74	2.00	2.37	1.90	2.24	2.24	2.34	2.06	2.61	3.45

Instance	S	E	H	L	SE	SH	SL	ME	MH	ML	LH	LL	LE
sprint_hidden6	16.17	14.09	15.77	11.68	15.37	14.18	11.74	16.66	14.98	13.72	12.06	13.35	19.07
sprint_hidden7	11.14	16.26	12.06	13.69	17.88	16.87	11.59	12.74	11.69	13.10	18.04	17.75	25.85
sprint_hidden8	12.42	14.37	14.35	20.06	13.56	14.67	11.76	16.38	11.28	13.97	12.99	15.45	19.80
sprint_hidden9	8.01	10.41	7.14	9.75	7.08	10.93	9.88	12.00	8.65	11.14	10.95	11.89	14.86
sprint_hidden10	8.88	16.32	10.93	12.23	15.51	14.52	14.16	14.44	15.75	13.97	15.84	16.61	20.96
medium_early1	1.02	0.90	1.08	0.93	0.77	3.87	0.90	0.86	0.97	0.94	2.92	0.97	0.99
medium_early2	2.98	0.78	0.90	0.76	0.87	4.46	0.85	1.89	0.70	0.99	3.37	1.03	1.37
medium_early3	3.15	0.64	0.92	0.76	1.07	1.07	1.27	0.93	1.03	0.94	3.02	1.21	1.32
medium_early4	2.83	1.08	0.88	0.94	1.04	0.91	0.89	2.05	0.86	1.31	3.26	1.52	1.23
medium_early5	4.15	1.08	1.09	1.07	0.85	4.56	1.05	1.11	0.81	1.35	3.33	1.11	1.60
medium_late1	7.24	10.44	6.99	8.34	6.34	9.50	7.25	6.95	8.01	8.61	8.50	7.21	9.75
medium_late2	3.31	3.39	3.28	2.67	3.31	2.01	3.15	3.62	3.02	2.79	2.60	2.96	4.60
medium_late3	3.09	3.10	3.38	3.34	3.43	2.92	2.74	6.05	2.94	2.87	3.10	4.70	4.48
medium_late4	2.55	2.26	2.13	2.46	2.60	2.61	2.23	6.84	1.81	2.11	2.43	2.85	3.13
medium_late5	10.25	11.07	10.16	11.85	11.63	13.83	11.49	27.08	10.04	8.94	12.22	13.03	17.95
medium_hidden1	11.95	16.08	11.03	13.50	13.63	15.23	14.32	14.26	14.15	12.05	14.83	18.27	17.28
medium_hidden2	17.43	16.66	12.70	13.41	14.11	16.91	17.05	27.85	15.99	15.91	19.30	16.50	21.63
medium_hidden3	2.46	2.50	2.87	3.49	2.80	2.98	2.95	2.66	3.22	3.11	7.67	3.40	4.34
medium_hidden4	3.21	3.43	3.29	4.22	3.59	4.38	4.52	4.90	3.13	4.41	8.67	3.88	5.39
medium_hidden5	9.20	13.78	8.73	12.27	9.96	8.82	9.51	8.12	14.40	8.52	25.41	10.85	19.98
long_early1	0.35	0.25	0.41	0.68	0.18	0.81	0.89	0.00	0.41	0.78	0.73	0.90	0.57
long_early2	11.19	1.23	1.04	1.20	1.07	1.90	1.55	7.00	1.00	1.53	1.68	1.56	1.61
long_early3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.77	0.00	0.00	2.28	0.00	0.00
long_early4	7.58	0.25	0.00	0.00	0.25	0.78	0.76	0.25	0.18	0.66	5.29	0.93	1.07
long_early5	7.81	0.47	0.00	0.18	0.00	0.70	0.57	0.00	0.00	0.57	0.52	1.01	0.79
long_late1	8.45	10.15	8.35	10.15	9.14	8.35	8.12	8.02	10.59	9.78	11.16	8.97	14.72
long_late2	10.61	14.57	9.90	10.41	10.01	10.15	10.71	7.23	9.25	13.47	10.54	11.84	13.32
long_late3	10.73	8.85	8.20	9.93	10.57	9.44	8.56	8.58	9.90	7.57	10.53	12.38	16.33
long_late4	8.45	9.35	11.62	11.63	9.66	12.55	9.53	9.64	8.39	10.50	10.79	13.41	17.29
long_late5	6.29	8.45	8.11	7.80	8.48	8.57	8.33	6.84	8.19	8.64	9.12	9.67	14.99
long_hidden1	17.56	15.22	11.97	20.77	14.95	16.36	15.98	76.19	11.20	11.87	15.84	15.28	26.78
long_hidden2	2.53	3.95	2.28	2.93	2.04	3.10	2.95	17.02	2.67	3.22	3.34	3.27	5.14
long_hidden3	3.30	4.81	3.81	3.95	4.15	4.14	3.51	3.28	3.14	4.07	4.10	3.80	5.39
long_hidden4	4.18	4.55	3.71	5.05	3.24	3.49	3.15	2.73	3.45	3.05	16.36	5.10	5.35
long_hidden5	4.03	6.22	5.18	4.66	4.60	5.13	3.91	5.72	6.28	5.54	5.51	5.17	6.57