A Genetic Programming Hyper-Heuristic Approach to Automated Packing

Matthew Hyde, BSc

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy

March 2010

Abstract

This thesis presents a programme of research which investigated a genetic programming hyper-heuristic methodology to automate the heuristic design process for one, two and three dimensional packing problems.

Traditionally, heuristic search methodologies operate on a space of potential solutions to a problem. In contrast, a hyper-heuristic is a heuristic which searches a space of heuristics, rather than a solution space directly. The majority of hyper-heuristic research papers, so far, have involved selecting a heuristic, or sequence of heuristics, from a set predefined by the practitioner. Less well studied are hyper-heuristics which can create new heuristics, from a set of potential components.

This thesis presents a genetic programming hyper-heuristic which makes it possible to automatically generate heuristics for a wide variety of packing problems. The genetic programming algorithm creates heuristics by intelligently combining components. The evolved heuristics are shown to be highly competitive with human created heuristics. The methodology is first applied to one dimensional bin packing, where the evolved heuristics are analysed to determine their quality, specialisation, robustness, and scalability. Importantly, it is shown that these heuristics are able to be reused on unseen problems. The methodology is then applied to the two dimensional packing problem to determine if automatic heuristic generation is possible for this domain. The three dimensional bin packing and knapsack problems are then addressed. It is shown that the genetic programming hyper-heuristic methodology can evolve human competitive heuristics, for the one, two, and three dimensional cases of both of these problems. No change of parameters or code is required between runs. This represents the first packing algorithm in the literature able to claim human competitive results in such a wide variety of packing domains.

Acknowledgements

My sincere thanks go to my two supervisors, Professor Edmund Burke, and Professor Graham Kendall, for the encouragement and flexibility they have given to me over the last three years. Through their support and trust, I have had the opportunity to direct my part in an interesting and very challenging project, and to enjoy myself along the way.

Secondly, I have no doubt that this thesis is stronger because of the working relationship I have had with Dr John Woodward, on the Engineering and Physical Sciences Research Council (EPSRC) project that has funded this PhD. I greatly valued our meetings, and his advice. I would like to also thank Dr. Gabriela Ochoa, for her support in the final stages of the PhD, especially helping me to ensure that I have included all the relevant literature.

My thanks also go to the EPSRC, whose funding made this project possible. In addition, many thanks go to the rest of the academic and administrative staff at the Automated Scheduling, optimisAtion and Planning (ASAP) research group. I cannot imagine a more interesting, diverse, and friendly group of people to work alongside, and I have learned a lot from all of you.

Contents

List of Figures viii

List of Tables xii

1 Introduction 1
- 1.1 Structure of Thesis . . . 6
- 1.2 Academic Publications Produced . . . 9

2 Literature Review 11
- 2.1 Introduction . . . 11
- 2.2 Hyper-Heuristics . . . 12
  - 2.2.1 Hyper-Heuristics to Choose Heuristics. . . . 14
  - 2.2.2 Hyper-Heuristics to Create Heuristics. . . . 21
- 2.3 One Dimensional Packing . . . 25
  - 2.3.1 Problem Definitions . . . 25
  - 2.3.2 Previous Work . . . 26
- 2.4 Two Dimensional Packing . . . 30
  - 2.4.1 Problem Definitions . . . 30
  - 2.4.2 Exact Methods . . . 32
  - 2.4.3 Heuristic Methods . . . 33
  - 2.4.4 Metaheuristic Methods . . . 34
- 2.5 Three Dimensional Packing . . . 36
  - 2.5.1 Problem Definitions . . . 36
  - 2.5.2 Exact Methods . . . 37
  - 2.5.3 Heuristic and Metaheuristic Methods . . . 38
- 2.6 Complexity Status . . . 40
- 2.7 Genetic Programming . . . 41
  - 2.7.1 Program Structure . . . 42
  - 2.7.2 Initialisation . . . 42
  - 2.7.3 The Five Preparatory Steps . . . 43
  - 2.7.4 Genetic Operators . . . 45
  - 2.7.5 Theoretical Concepts . . . 48
  - 2.7.6 Achievements of Genetic Programming . . . 49
  - 2.7.7 Summary . . . 50

2.8 Conclusion 50

3 Evolving Heuristics for the One Dimensional Bin Packing Problem 52
3.1 Introduction 52
3.2 The First-Fit Heuristic 53
3.3 Methodology 54
3.3.1 How The Heuristic is Applied 54
3.3.2 Structure of the Heuristics 56
3.3.3 Simple Example of a Heuristic Constructing a Solution 57
3.3.4 Heuristics Producing Illegal Solutions 59
3.3.5 How the Heuristics are Evolved 61
3.4 Results 65
3.4.1 Individual Results 65
3.4.2 Statistical Tests 69
3.4.3 Comparison with First-Fit and Best-Fit, Employing the Basic Fitness Function 70
3.4.4 Comparison with First-Fit and Best-Fit, Employing the Second Fitness Function 72
3.4.5 Comparison with First-Fit and Best-Fit, Removing the $\leq$ Function . 74
3.5 Conclusion 76

4 The Reusability of Evolved Online Bin Packing Heuristics 79
4.1 Introduction 79
4.2 The Best-Fit Heuristic 80
4.3 Methodology 82
4.3.1 How The Heuristic is Applied 82
4.3.2 Structure of the Heuristics 84
4.3.3 Simple Example of a Heuristic Constructing a Solution 85
4.3.4 How the Heuristics are Evolved 87
4.4 Results 90
4.4.1 Example of an Evolved Heuristic 90
4.4.2 The Quality of Evolved Heuristics 92
4.4.3 Specialisation of Evolved Heuristics 93
4.4.4 Robustness of Evolved Heuristics 95
4.5 Conclusion 97

5 The Scalability of Evolved Online Bin Packing Heuristics 99
5.1 Introduction 99
5.2 Methodology 100
5.2.1 How the Heuristic is Applied 100
5.2.2 Structure of the Heuristics 102
5.2.3 How the Heuristics are Evolved 104
5.3 Results 106
5.3.1 Performance past the training range 107
5.3.2 Performance over the training range 109

5.3.3 Two Evolved Heuristics . . . 111
5.4 Conclusion . . . 117

6 Evolving Heuristics for Two Dimensional Strip Packing 120
6.1 Introduction . . . 120
6.2 Representation . . . 122
6.3 Methodology . . . 125
6.3.1 How the Heuristic is Applied . . . 125
6.3.2 The Structure of the Heuristics . . . 127
6.3.3 Example of the Packing Process . . . 130
6.3.4 How the Heuristics are Evolved . . . 134
6.4 Results . . . 136
6.4.1 Explanation of Benchmark Algorithms . . . 136
6.4.2 Comparison with Metaheuristics and the Best-Fit Heuristic . . . 139
6.4.3 Comparison with Reactive GRASP . . . 139
6.4.4 Example Packings . . . 140
6.4.5 Results with Parsimony Pressure and Cache of Heuristic Evaluations 142
6.5 Conclusion . . . 144

7 A Hyper-Heuristic for 1D, 2D and 3D Packing Problems 147
7.1 Introduction . . . 147
7.2 Problem Descriptions . . . 149
7.2.1 The Knapsack Problem . . . 149
7.2.2 Knapsack Packing in Two and Three Dimensions . . . 149
7.2.3 The Bin Packing Problem . . . 150
7.2.4 Bin Packing in Two and Three Dimensions . . . 150
7.3 Representation . . . 151
7.3.1 Bin and Corner Objects . . . 151
7.3.2 Valid Placement of Pieces . . . 153
7.3.3 Extending a Corner's Surfaces . . . 153
7.3.4 Filler Pieces . . . 154
7.3.5 Bin Packing, and Packing in Lower Dimensions . . . 155
7.4 Methodology . . . 156
7.4.1 How the Heuristic is Applied . . . 156
7.4.2 The Structure of the Heuristics . . . 159
7.4.3 How the Heuristics are Evolved . . . 160
7.5 The Data Sets . . . 162
7.5.1 One Dimensional Instances . . . 163
7.5.2 Two Dimensional Instances . . . 164
7.5.3 Three Dimensional Instances . . . 165
7.6 Results . . . 166
7.6.1 An Example Calculation . . . 167
7.6.2 Bin Packing Results . . . 169
7.6.3 Knapsack Results . . . 171
7.7 Conclusion . . . 172

8 Conclusion **175**
8.1 Context 175
8.2 Summary of Work 177
8.2.1 Chapters 3 - 5 177
8.2.2 Chapter 6 178
8.2.3 Chapter 7 178
8.3 Extensions and Future Work 180
8.3.1 Evolving Reusable Heuristics for Two and Three Dimensional Packing 180
8.3.2 Specialisation and Robustness of Two Dimensional Packing Heuristics 181
8.3.3 Intelligently Select which Allocations to Consider 181
8.3.4 Improve the Three Dimensional Packing Representation 181
8.3.5 Employ Higher Level Functions and Terminals 182
8.3.6 Evolving one General Packing Heuristic 182
8.4 Final Remarks 183

References **184**

List of Figures

1.1 A diagram which clarifies one of the key themes of this thesis. One of the motivations for hyper-heuristic research is to automate the heuristic design process. This thesis presents methodologies which allow the human to be replaced by genetic programming at the algorithm design stage, while currently a human must still specify the potential components of the heuristic to be built. Such an automated system means that human involvement in heuristic design can stop earlier in the process. 3
1.2 A conceptual graph of the potential advantage of hyper-heuristic systems which generate heuristics. A metaheuristic system can begin to obtain solutions to instances immediately. The hyper-heuristic cannot during the evolution, but once a heuristic is evolved it can obtain solutions to problem instances quickly. In the long run, the hyper-heuristic system as a whole requires less time because it produces one quick constructive heuristic. . . . 5
1.3 The structure of the thesis over the chapters 3 through to 8. The diagram shows that the thesis progresses from evolving non-reusable heuristics, to evolving reusable heuristics, and then back again to systems which evolve one heuristic per instance. The diagram also shows which chapters relate to which problem domains, and that chapter 7 presents results on all three domains 7

2.1 The relationship between a hyper-heuristic, the domain specific heuristics, and a problem instance [252] 15
2.2 A diagram showing where the heuristics from table 2.1 would place a piece of size 60. Next-fit only has access to the highest indexed bin and an empty bin, and it chooses the empty bin in this case because the piece does not fit into the highest indexed bin 28
2.3 An example tree structure representing a program that calculates the space left in the bin in figure 2.4 43
2.4 The features of a one dimensional bin to which the tree in figure 2.3 could be applied to 43
2.5 An example of a genetic programming crossover operation being performed on two parent trees to create two child trees 47

2.6 An example of a genetic programming mutation operation being performed. Only one parent tree is necessary, and one child tree is produced. The subtree rooted at the circled node is deleted, and a new subtree is grown at the same point, by randomly selecting each node 48

3.1 Pieces remaining to be packed, and a partial solution 54
3.2 The first piece is put into the third bin because it is the first bin with enough space to accommodate it 54
3.3 The second piece is put into the fourth bin 54
3.4 The final two pieces are put into the first and fifth bins 54
3.5 An example heuristic used in the analysis in section 3.3.3, to pack the pieces in figures 3.6 to 3.9 57
3.6 A partial solution. We will use the heuristic in figure 3.5 to pack the four remaining pieces 58
3.7 The piece of size 70 is packed into the first bin for which the heuristic returns a positive number, the fifth bin 58
3.8 The piece of size 85 is packed into the sixth bin 58
3.9 The piece of size 30 is packed into the third bin 58
3.10 A heuristic that puts every piece in the first bin 60
3.11 A heuristic that puts no pieces in any bin 60
3.12 A heuristic that puts no pieces in any bin 60
3.13 Tree A 66
3.14 Tree B 66
3.15 Tree C 66
3.16 Tree D 66
3.17 Tree E 68
3.18 Tree F 68

4.1 The start of the example of section 4.2, the piece of size 70 will be packed next by the best-fit heuristic 81
4.2 The piece of size 70 is packed into the fourth bin because it has the least free space of all the bins that the piece fits into 81
4.3 The next piece is packed into the third bin after all the bins are considered 81
4.4 The final two pieces are packed into the second and first bins 81
4.5 An example heuristic, used in the example packing described in section 4.3.3 and figures 4.6 to 4.9 85
4.6 The initial partial solution from section 4.3.3. The heuristic in figure 4.5 will pack the first three pieces. 86
4.7 All of the bins are scored by the heuristic, the bin on the far right scores the highest with 34500. A new empty bin is opened because the existing one receives the piece. 86
4.8 The newly opened bin on the far right scores the highest with 32250. Again, a new empty bin is opened because there must always be one available. . . 86
4.9 For the third time, the empty bin receives the piece. The heuristic appears to prefer to place each piece into a new bin. 86

4.10 Diagram of the piece size distributions of each class of instances created for the experiments of chapter 4 89
4.11 An example heuristic, evolved on T_{10-90} 91

5.1 The results for H_{100}, H_{250}, and H_{500} on the validation instances. The vertical axis shows the amount of bins by which the 30 heuristics from each group beat best-fit on average. The horizontal axis shows the number of pieces packed. The results shown are the average amount of bins required by the heuristics over the 20 validation instances. 107
5.2 The results for the heuristics evolved on 100, 250, and 500 pieces of the training instances, on the first 500 pieces of each of the 20 validation instances. This is a 'zoomed in' view of the same data displayed in figure 5.1. The vertical axis shows the amount of bins by which the 30 heuristics from each group beat best-fit on average. The horizontal axis shows the number of pieces packed. The results shown are the average results of the 30 heuristics over the 20 instances. 109
5.3 The function $C/(E-S)$, which represents the best-fit heuristic when applied in the fixed framework used in this chapter. The defining feature is the 'wall' which exists on the boundary of where the piece size matches the 'emptiness' of the bin. Behind this wall the piece size is larger than the emptiness, so the piece does not fit in the bin. The bin which receives the maximum score will be the bin with the least emptiness out of all of the bins that the piece fits into. 113
5.4 A heuristic requiring 41448.4 bins on average over the validation set, thus outperforming best-fit. The graph is in the same format as figure 5.3, but rotated to put the 'high emptiness' side in the foreground. The heuristic has much the same structure as best-fit, however the corner curls up at the end as the piece size gets small and the emptiness gets large. There is also a 'ridge' in the landscape running in front of the wall, where certain combinations of piece size and emptiness are rated higher than those adjacent to the ridge on both sides. 114
5.5 A heuristic requiring 40573.3 bins on average over the validation set, thus outperforming best-fit. This is the same heuristic as is displayed in figure 5.6 115
5.6 A heuristic requiring 40573.3 bins on average over the validation set. This is the same heuristic as is displayed in figure 5.5 116

6.1 Bin structure after two pieces have been packed 123
6.2 The middle bin of figure 1 has been replaced by two new bins due to the third piece being packed into it 123
6.3 As no piece can fit into bin three of figure 2, the base of bin three has been raised, to combine it with bin two 123
6.4 When the fourth piece is packed into the second bin, the heights of bin two and three are the same and they have been combined 123
6.5 An example heuristic, which is analysed in section 6.3.3 by packing the pieces of figure 6.6 130

6.6 The pieces that the heuristic in figure 6.5 will pack for the example described in section 6.3.3 . . . 131
6.7 The current partial solution, the heuristic will choose a piece from figure 6.6 to be placed in this solution . . . 131
6.8 All of the locations where piece one from figure 6.6 can be placed . . . 131
6.9 Allocation A from figure 6.8 in detail . . . 131
6.10 Allocation C from figure 6.8 in detail . . . 131
6.11 Both of the potential allocations for piece two . . . 133
6.12 The new partial solution after the allocation which received the highest score has been performed . . . 133
6.13 Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 64 . . . 141
6.14 Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 256 . . . 141
6.15 The heuristic that packed figure 6.13 . . . 141
6.16 The heuristic that packed figure 6.14 . . . 141

7.1 An initialised bin with one 'corner' in the back-left-bottom corner of the bin 152
7.2 A bin with one piece placed in the corner from figure 7.1 . . . 152
7.3 The three surfaces defined by the corner that is in the Y direction of the piece 152
7.4 The three surfaces defined by the corner that is in the X direction of the piece 152
7.5 An invalid placement of a new piece, because it exceeds the limit of the corner's XZ surface in the Z direction . . . 153
7.6 An invalid placement of a new piece, because it exceeds the limit of the corner's YZ surface in the Z direction . . . 153
7.7 A piece which reaches the limit of two surfaces of the corner that it was put into . . . 154
7.8 The three surfaces of the corner with the smallest available area . . . 155
7.9 A filler piece is put into the corner from 7.8, the surfaces of two corners are updated . . . 155

List of Tables

1.1 A summary of the implementation details for the genetic programming system presented in each chapter. The parameters subtly change over time due to experience, experimentation, and the capabilities of freely available genetic programming code 8

2.1 A description of existing constructive heuristics for one dimensional bin packing 27

3.1 The functions and terminals used in chapter 3, and descriptions of the values they return 56
3.2 Genetic programming initialisation parameters for chapter 3 61
3.3 The performance of 20 heuristics, each evolved on all 20 instances using the basic fitness function which is shown in equation 3.3, section 3.3.5 71
3.4 The performance of 20 heuristics each evolved on one instance, using the basic fitness function which is shown in equation 3.3, section 3.3.5 72
3.5 The performance of 20 heuristics each evolved on all 20 instances using the second fitness function which is shown in equation 3.4, section 3.3.5 73
3.6 The performance of 20 heuristics each evolved on one instance, using the second fitness function which is shown in equation 3.4, section 3.3.5 74
3.7 The performance of 20 heuristics each evolved on all 20 instances, without the $\leq$ function included in the function set, and with the second fitness function, which is shown in equation 3.4, section 3.3.5 75
3.8 The performance of 20 heuristics each evolved on one instance, without the $\leq$ function included in the function set, and with the second fitness function, which is shown in equation 3.4, section 3.3.5 77

4.1 The functions and terminals used in chapter 4, and descriptions of the values they return 85
4.2 Genetic programming initialisation parameters for chapter 4 88
4.3 Summary of the results obtained by the heuristic shown in figure 4.11 compared to the results of best-fit. The result shown is the total bins used over the 20 instances. 92
4.4 The performance of each set of evolved heuristics tested against best-fit . . 93

4.5 A comparison of the performance of six sets of evolved heuristics with the heuristic sets evolved on their respective super-class and super-super-class. The entries in the table state whether the set is better, worse, or the same as its super-class (or super-super-class) set. The values in brackets represent the probabilities that the performance values of both heuristic sets come from underlying populations with the same mean. For example, row 4 (labelled H_{10-29}) shows that the H_{10-29} set is better on the V_{10-29} instances than the set evolved on its super-class (H_{10-49}) and the set evolved on its super-super-class (H_{10-89}) 94
4.6 Summary of the illegal results when heuristic sets are each tested on three unrelated validation sets, containing piece sizes they did not pack in their evolution. A cross represents at least one illegal result, a circle represents no illegal results 96

5.1 The functions and terminals used in chapter 5, and descriptions of the values they return 102
5.2 The values returned by the standard protected divide function and the modified protected divide used in the experiments of this chapter. 103
5.3 Genetic programming initialisation parameters for the experiments of chapter 5 105
5.4 The number of bins used on average by the three groups of 30 evolved heuristics over the set of problems with 100,000 pieces, compared to the number used by best-fit 108

6.1 The functions and terminals used in chapter 6, and descriptions of the values they return 128
6.2 The values returned by the standard protected divide function, the modified protected divide used in the previous chapter, and the further modified function used in this chapter. 129
6.3 Initialisation parameters of the experiments of chapter 6 134
6.4 The benchmark problem instances used in this chapter 136
6.5 Results of our evolved heuristics compared to recent metaheuristic and constructive heuristic approaches, and the reactive GRASP approach. The results are highlighted, in bold type, where the hyper-heuristic system finds a heuristic which has equal or better results than the metaheuristic approaches and the best-fit algorithm for that instance. The GRASP algorithm is generally the best approach, the bold values are intended to show how the evolved heuristics compare to the other approaches 137
6.6 Difference in performance of the best evolved heuristics for each instance and the reactive GRASP, the figures represent the extent to which the GRASP performs better than the evolved heuristics. 140
6.7 A comparison of the time taken to evolve the heuristics using the previous implementation, and the improved implementation. 143

7.1 The functions and terminals used in chapter 7, and descriptions of the values they return 159

7.2 Initialisation parameters of the experiments of chapter 7 161
7.3 A summary of the 18 data sets used in the experiments of this chapter . . . 163
7.4 The individual instance results for the Thpack9 bin packing data set 168
7.5 Summary of bin packing results, and the percentage improvement over the best results in the literature 169
7.6 Summary of knapsack results, and the percentage improvement over the best results in the literature 170

Chapter 1

Introduction

Many combinatorial problems of practical interest are intractable, which, put quite simply, means that the search space they create is too large to search exhaustively. For this reason, heuristics are created to obtain reasonably good solutions in a reasonable amount of time. Heuristics provide no performance guarantees, but, when well designed and implemented, they are methods which are generally recognised as able to produce good solutions most of the time. The research into metaheuristics, over the last few decades, has shown that more advanced search technology is often required to obtain the high levels of solution quality. The term metaheuristic refers to a class of heuristic methods [50]. The term was first used by Glover in [122] to describe tabu search, and is subsequently defined in 1997 as "a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality" [123]. In other words, they utilise various methods to avoid the search being drawn into local optima at the expense of finding the global optimum. Genetic algorithms, simulated annealing, and tabu-search are three well known examples of metaheuristics.

To develop a (meta)heuristic search system, one could analyse the problem and choose the best search paradigm to use from the literature. Another option would be to utilise knowledge of the structure of the problem to create a new heuristic. Often, it is not obvious which heuristic to choose because of either the complexity of the problem, or the lack of knowledge of the circumstances in which the existing heuristics are effective.

In addition to the difficulty of selecting an appropriate heuristic, a (meta)heuristic implementation often cannot be applied to problem domains other than those which they are specifically created for, or even to different instances of the same problem [47]. For example, a (meta)heuristic methodology developed for the university course timetabling

problem may not perform well on the exam timetabling problem. Furthermore, a system developed for the course timetabling problem in one university, may not work well for the same problem in a different university. There are features of the problems (and problem instances) that make them fundamentally different from each other, such as the number of students, the size of rooms and other available resources. The methodologies effective for one set of constraints may not be effective when the constraints are changed. In other words, the implementation of these search methodologies is often bespoke in nature, and this makes them potentially expensive systems, as they cannot be reused once developed for a given application. The relative expense of bespoke systems, and the difficulty involved in the selection of an appropriate heuristic, both serve as motivations for hyper-heuristic research.

A hyper-heuristic can be defined as a heuristic which searches a space of heuristics, rather than a solution space directly [47, 52, 74, 239]. In other words, it is a heuristic which chooses between a set of heuristics. The aim is to automate the decision of which heuristic(s) to use for a given problem.

The term hyper-heuristic has been coined relatively recently [47, 52, 74, 252]. However, the first research into methods which search a heuristic space dates back to the early 1960s, when Fisher and Thompson developed a hyper-heuristic method for the job shop scheduling problem [106, 107]. At every decision point, one of the heuristics was applied to the current solution based on a dynamic weighting. The weighting was increased if the heuristic obtained a better solution, and decreased if not. Therefore, if a heuristic produced an improvement, it was more likely to be chosen again. This hyper-heuristic was found to be better than applying a randomly chosen heuristic. Also, randomly choosing a sequence of heuristics was found to obtain better results than any one heuristic could achieve alone, providing early evidence that using combinations of existing heuristics during the search could be beneficial.

Recently, many metaheuristics have been employed as hyper-heuristics. For example, simulated annealing [87], genetic algorithms [77, 134, 132, 133, 241, 263], tabu search [52], and an ant algorithm [51]. These methods all require a set of heuristics to choose from, which must be supplied by the practitioner. However, if the problem is new, there may be no pre-existing heuristics available, or the existing heuristics may be sub-standard. Even if there are heuristics available, better quality heuristics may be hard to discover manually if they are counterintuitive in nature. These issues and circumstances serve as the motivation

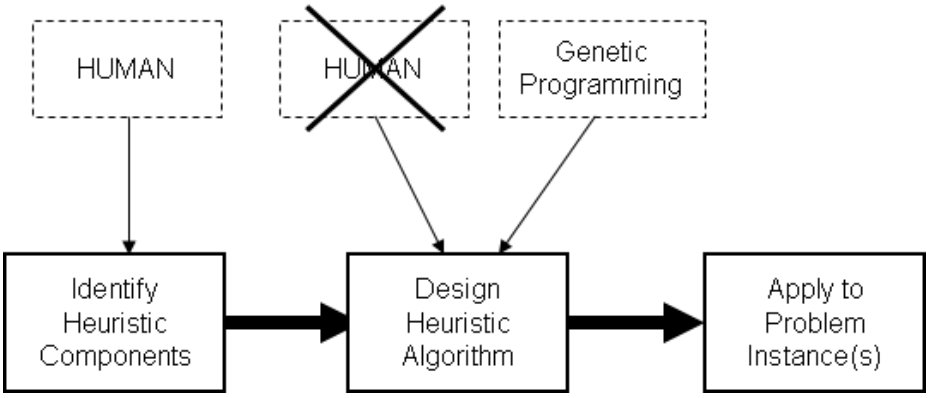

FIGURE 1.1: A diagram which clarifies one of the key themes of this thesis. One of the motivations for hyper-heuristic research is to automate the heuristic design process. This thesis presents methodologies which allow the human to be replaced by genetic programming at the algorithm design stage, while currently a human must still specify the potential components of the heuristic to be built. Such an automated system means that human involvement in heuristic design can stop earlier in the process.

for research into hyper-heuristic systems which *create* heuristics, rather than select between pre-existing heuristics.

The subject of this thesis is a hyper-heuristic system which automatically designs heuristics, using a genetic programming algorithm. A set of potential components must be defined by the user, and the system searches the space of all heuristics that can be created from the given components. While the specification of the components is not automated, it is a process which requires less human input than to specify a set of fully functional heuristics. Fukunaga states that humans excel at identifying good potential components of methods to solve problems, but combining them seems to be a more difficult undertaking [112]. This concept is central to the motivation for the work in this thesis, and is depicted graphically in figure 1.1. As problems in the real world become more complex, identifying ways to automate this process may become fundamental to the design of heuristics, because it will become more difficult to manually combine their potential components in ways that fully exploit the structure of a complex problem.

There are a number of advantages of such an approach. There is a possibility of discovering new heuristics which are unlikely to be invented by a human analyst, due to their counterintuitive nature. Another advantage is that a different heuristic can be created for each individual instance, meaning that the result obtained on each is more likely to be

better than that obtained by one general heuristic. Human created heuristics are designed to perform well over many problems, because it would take too long to manually develop a new heuristic for every individual instance. This thesis will show examples of counterintuitive heuristics, which nevertheless perform better than those created by humans. The results also show that a heuristic can indeed be automatically created for each instance, meaning that the results of the hyper-heuristic as a whole are mostly superior to those of a comparable human created heuristic.

The heuristics can be seen as 'disposable', if they are created to obtain a result for one instance. In this case, the hyper-heuristic is searching the heuristic space to indirectly find a good solution in the solution space. However, the process outputs a heuristic as well as a solution, and this heuristic could be reused on new problems. Therefore, this thesis also investigates the circumstances in which it is possible to apply the automatically created heuristic to unseen problems and still obtain good results.

Evolving a disposable heuristic can take in the order of minutes or hours depending on the size of the problem instance being solved. Every evaluation of an individual in the population requires the training instance to be solved by the individual. A fitness is assigned to the individual based on the result. If the problem being solved is large then this increases the total time required for the evolution. It would be quicker to apply a one-pass constructive heuristic, but its result is generally worse than the result of a heuristic which has been evolved on the instance. Also, the time to apply a metaheuristic, searching the space of solutions, would be of the same order of magnitude as applying the hyper-heuristic to search the space of heuristics.

If the system is capable of evolving heuristics that can be reused on new problem instances, the run time issues change. One motivation for hyper-heuristic research is that once a heuristic is evolved, and specialised to a class of instances, it can obtain solutions to new instances very quickly. Examples of this will be shown in this thesis. The time necessary for evolution of a heuristic, means that in the short term it can be more efficient to run a metaheuristic search. However, if many problems must be solved, then the total time taken for the metaheuristic runs will soon become larger than the time to evolve a quick heuristic and then run it multiple times. This is shown graphically in figure 1.2.

A key concept is that one-pass constructive heuristics 'tailored' to a class of problem instances can obtain better results, more quickly, than more complex metaheuristic algorithms. Historically, one-pass constructive heuristics in the literature are intended to

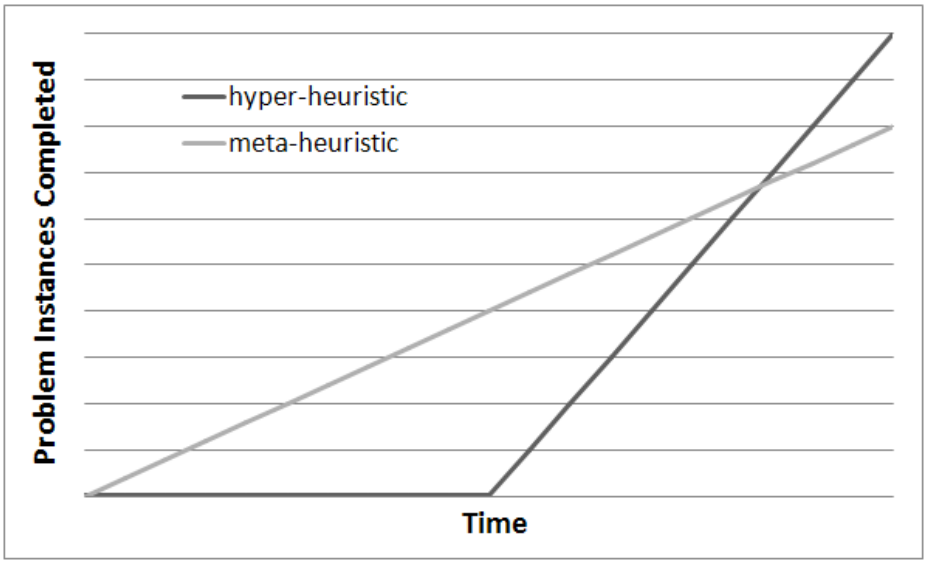

FIGURE 1.2: A conceptual graph of the potential advantage of hyper-heuristic systems which generate heuristics. A metaheuristic system can begin to obtain solutions to instances immediately. The hyper-heuristic cannot during the evolution, but once a heuristic is evolved it can obtain solutions to problem instances quickly. In the long run, the hyper-heuristic system as a whole requires less time because it produces one quick constructive heuristic.

be used on any problem instance. The literature contains many worst case and average case performance analyses for these heuristics over all problems. However, in many cases, a practitioner is interested in the performance on their special case, which will contain a certain structure that a bespoke heuristic can exploit. The focus on general heuristics could be because a bespoke heuristic has little value outside of the problem set it is designed for, and therefore it is expensive to design. Research into hyper-heuristics aims to reduce that cost by automating the design process.

The domain of cutting and packing will be used in order to demonstrate the potential of genetic programming to automatically create heuristics. Cutting and packing problems involve assigning a number of small items to one or more larger items. This can represent situations where shapes must be cut from larger stock sheets efficiently, or where boxes must be packed into larger containers while minimising waste. Heuristics for this domain are well studied, both empirically and theoretically, and the literature shows that a vast array of heuristics, metaheuristics, and exact methods have been created and analysed. This thesis shows that this process of heuristic creation can be automated.

While not being a defining characteristic, the issue of generality is central to hyper-heuristic research. Generality, in this case, means the ability to successfully apply the system to a number of problem domains. To show the generality of this genetic programming system, we will use the different problem domains of knapsack packing and bin packing. A knapsack problem is one where a subset of the set of small items must be chosen to fit into one larger item. Each small item has a value associated with it, as well as a size. The objective is to find the subset which has the maximum total value, while still fitting into the larger item. A bin packing problem involves assigning all of the small items, each to one of a number of large items. The objective is to minimise the number of large items needed to accommodate all of the small items.

The knapsack and bin packing problems can be defined in one, two, and three dimensions. In this thesis, the genetic programming system will be applied to problems of each dimensionality. Finally, the developed system will be shown to operate on both knapsack and bin packing problems, of any dimensionality, without any change in parameters. Indeed, it is the first system in the literature that is able to claim human competitive results in such a wide variety of packing domains.

In summary, this thesis will investigate the quality, specialisation, robustness, scalability, and potential for reuse of the automatically generated heuristics. It will be shown that they can be generated for a number of different problem domains, and that the results they obtain can be human competitive.

1.1 Structure of Thesis

The thesis is structured as follows. Chapter 2 presents a review of the relevant literature for genetic programming, hyper-heuristics, and one, two, and three dimensional packing. Chapters 3 to 5 present work on one dimensional bin packing. Heuristics are evolved in Chapter 6 for two dimensional strip packing. Chapter 7 presents the most recent work, a hyper-heuristic system which evolves heuristics for the one, two, or three dimensional bin packing or knapsack problem.

Figure 1.3 presents an overview of the thesis, showing the themes and problem domains addressed in each chapter. The diagram is intended to be read from the ‘start’ box to the ‘conclusions and future work’ box, following the arrows. The main results of each chapter are attached to each chapter’s box, and for chapters 3 to 5, the diagram shows

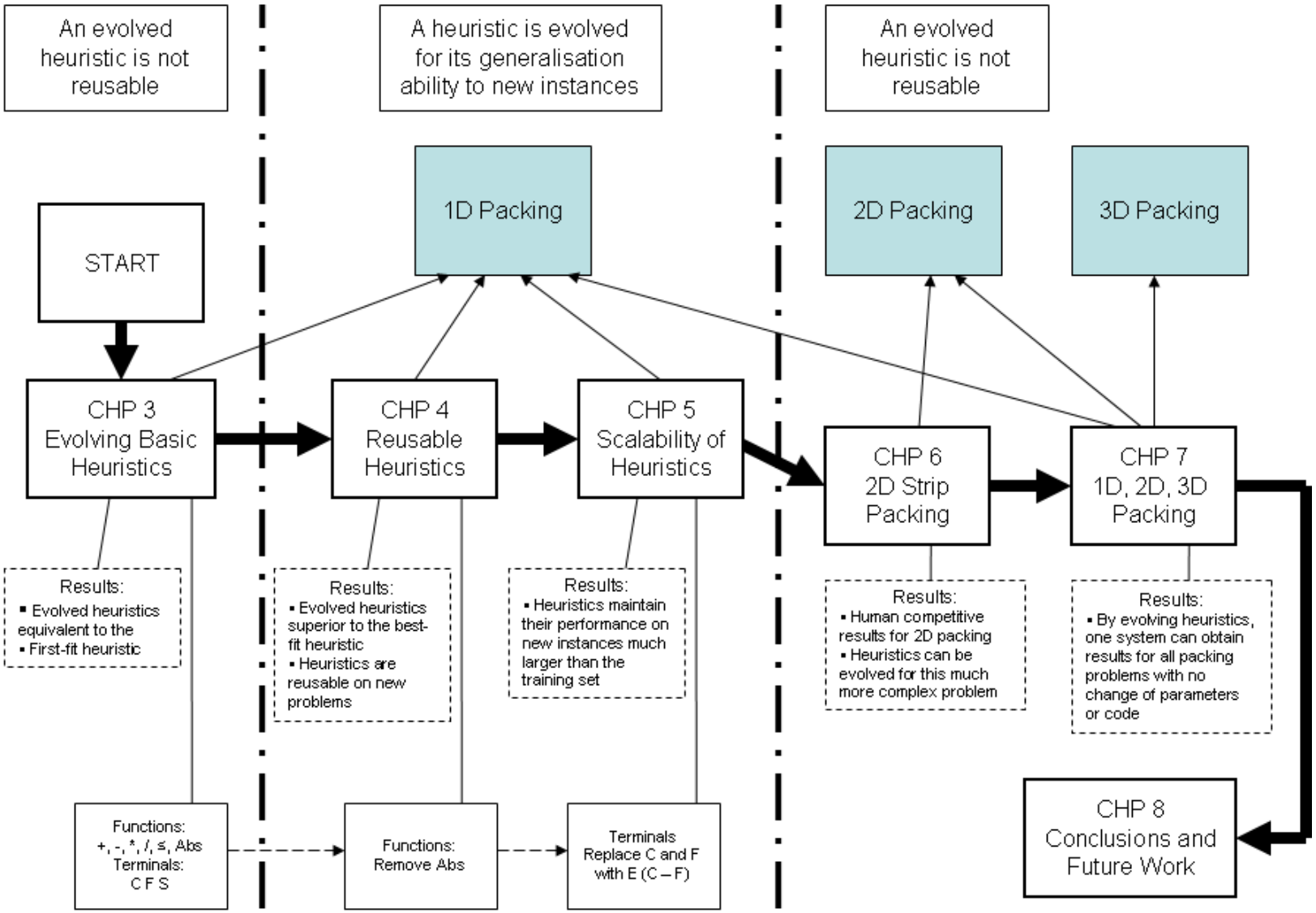

FIGURE 1.3: The structure of the thesis over the chapters 3 through to 8. The diagram shows that the thesis progresses from evolving non-reusable heuristics, to evolving reusable heuristics, and then back again to systems which evolve one heuristic per instance. The diagram also shows which chapters relate to which problem domains, and that chapter 7 presents results on all three domains

TABLE 1.1: A summary of the implementation details for the genetic programming system presented in each chapter. The parameters subtly change over time due to experience, experimentation, and the capabilities of freely available genetic programming code

Feature	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
GP Code	Author's Code		ECJ		
Population Size	1000		1024	64, 128, 256	1000
Generations	50				
Initialisation	Grow		Ramped half and half		
Protected Divide	$\frac{x}{0} = 1$		$\frac{x}{0} = \frac{x}{0.5}$	$\frac{x}{0} = \frac{x}{0.001}$	
Selection	Roulette Wheel		Tournament		
Crossover	90%			85%	
Mutation	0%			10%	
Reproduction	10%			5%	

how the genetic programming functions and terminals change as improvements to the one dimensional bin packing system were made. The diagram also displays the problem domains for which the heuristics in each chapter are evolved for. The diagram is partitioned into three sections, showing that the heuristics evolved in chapter 3 are not reusable, while in chapters 4 and 5 the thesis investigates how reusable the heuristics are after they have been evolved. In chapters 6 and 7, the focus returns to successfully evolving heuristics specialised to one problem instance, which enables the genetic programming system as a whole to achieve good results on a variety of problem domains.

The implementation details of the genetic programming system subtly change throughout the thesis, due to experience and experimentation. Table 1.1 gives a summary of the main changes that occur. Chapters 3 and 4 use genetic programming code written by the author, while the subsequent chapters use the freely available ECJ (Evolutionary Computation in Java) package, available at http://www.cs.gmu.edu/∼eclab/projects/ecj/.

1.2 Academic Publications Produced

The following academic papers and one book chapter have been produced as a result of this research. For each publication, the chapter number is given for where the work is presented in this thesis.

- Chapter 3

  E.K. Burke, M.R. Hyde and G. Kendall. 2006. "Evolving Bin Packing Heuristics with Genetic Programming", In Runarsson, Thomas., Beyer, Hans-Georg., Burke, Edmund., J.Merelo-Guervos, Juan., Whitley, Darrell., and Yao, Xin., eds.: LNCS 4193, *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006).* Reykjavik, Iceland. September, 2006. Springer Lecture Notes in Computer Science, Volume 4193. pp. 860–869.

- Chapter 4

  Burke E. K., Hyde M., Kendall G., and Woodward J. 2007. "Automatic Heuristic Generation with Genetic Programming: Evolving a Jack-of-all-Trades or a Master of One", In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO 2007).* London, UK. July 2007. pp. 1559–1565.

  Burke E. K., Hyde M., Kendall G., Ochoa G., Ozcan E., and Woodward J. 2009. "Exploring Hyper-heuristic Methodologies with Genetic Programming", In C. Mumford and L. Jain, eds.: *Computational Intelligence: Collaboration, Fusion and Emergence*, Intelligent Systems Reference Library. Springer. ISBN:978-3-642-01798-8.

- Chapter 5

  Burke E. K., Hyde M., Kendall G., and Woodward J. 2007. "Scalability of Evolved On Line Bin Packing Heuristics", In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007).* Singapore. September 2007. pp. 2530–2537.

- Chapter 6

  Burke E. K., Hyde M., Kendall G., and Woodward J. 2010. "A Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics". *IEEE Transactions on Evolutionary Computation.* Accepted, to Appear.

- Chapter 7

  Allen S., Burke E. K., Hyde M., and Kendall G. 2009. "Evolving Reusable 3D Packing Heuristics with Genetic Programming". In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO 2009)*. Montreal, Canada. July 2009. pp 931–938.

Chapter 2

Literature Review

2.1 Introduction

This chapter will provide a review of the relevant literature for the areas of work studied in this thesis. There are three main areas of work relevant to this thesis. They are hyper-heuristics, packing problems, and genetic programming.

The first area of work to be covered is hyper-heuristics. This term can be used to define a heuristic that searches a space of heuristics, rather than a space of solutions directly. Hyper-heuristics can be classified into two main types. The first type chooses one heuristic, or a sequence, from a pre-defined set. The second type automatically creates heuristics from a set of potential components. The hyper-heuristic developed for packing problems developed in this thesis falls into the second category.

The packing problems that we use to evaluate the hyper-heuristic are the one, two, and three dimensional bin packing problems, the two and three dimensional knapsack problems, and the two dimensional strip packing problem. In this chapter we review the previous work on packing problems by organising the subjects into three sections. Section 2.3 presents the one dimensional packing literature, section 2.4 presents two dimensional packing, and section 2.5 presents three dimensional packing.

Genetic programming is an evolutionary computation technique which evolves computer programs. This technique is employed in this thesis to evolve heuristics for the various packing problems outlined above. Section 2.7 is a brief introduction to genetic programming, providing an explanation of the main concepts. References to more detailed explanations are also provided.

The contribution of this thesis is to show it is possible for good packing heuris-

tics to be automatically generated by a genetic programming hyper-heuristic. The aim of this chapter is to put this contribution in context, and provide information about other approaches to the same problems.

2.2 Hyper-Heuristics

Hyper-heuristics can be defined as heuristics that search a space of heuristics, as opposed to searching a space of solutions directly [239]. The term "Hyper-Heuristics" is relatively new, however the basic idea has been around since the 1960's. For example, in 1961 (and again in 1963), Fisher and Thompson presented an algorithm which combines local job shop scheduling rules using a probabilistic learning technique [106, 107]. This can be classed as a hyper-heuristic because the learning algorithm chooses which of two heuristics to apply, and the chosen heuristic then selects the next job for the machine.

Research in this area is motivated by the "goal of raising the level of generality at which optimisation systems can operate" [47], and by the assertion that in many real-world problem domains, there are users who are interested in *"good-enough, soon-enough, cheap-enough"* solutions to their search problems, rather than *optimal* solutions [47]. In practice, this means researching systems that are capable of operating over a range of different problem instances and sometimes even across problem domains, without expensive manual parameter tuning, and while still maintaining a certain level of solution quality.

The 'No Free Lunch' theorem [278, 279] shows that all search algorithms have the same average performance over all possible discrete functions. This would suggest that it is not possible to develop a general search methodology for all optimisation problems as, over all possible discrete functions, "no heuristic search algorithm is better than random enumeration" [277]. However, it is important to recognise that this theorem is *not* saying that it is not possible to build search methodologies which are *more* general than is currently possible. It is often the case in practice that search algorithms are developed for a specific group of problems, for instance timetabling problems [243]. Often the algorithms are developed for a narrower set of problems within that group, for instance university course timetabling [54, 41] or exam timetabling problems [42]. Indeed, algorithms can be specialised further by developing them for a specific organisation, whose timetabling problem may have a structure very different to that of another organisation with different resources and constraints [7, 244]. At each of these levels, the use of domain knowledge can allow the

algorithms to exploit the structure of the set of problems in question. This information can be used to intelligently guide a heuristic search.

In the majority of cases, humans develop heuristics which exploit certain features of a problem domain, and this allows the heuristics to perform better on average than random search. Hyper-heuristic research is concerned with building systems which can automatically exploit the structure of a problem they are presented with, and create new heuristics for that problem, or intelligently choose from a set of pre-defined heuristics. In other words, hyper-heuristic research aims to automate the heuristic design process, or automate the decision of which heuristics to employ for a new problem.

The advantage of an automated heuristic design process, is in making optimisation tools and decision support available to organisations who currently solve their problems by hand, without the aid of computers. Examples of such organisations could be, for example, a primary school with a timetabling problem, or a small delivery company with a routing problem. It is often prohibitively expensive for them to employ a team of analysts to build a bespoke heuristic, which would be specialised to their organisation's problem. A more general system which automatically creates heuristics would be applicable to a range of organisations, potentially lowering the cost to each. It may be that there is a trade-off between the generality of such a system, and the quality of the solutions it obtains. However, organisations for whom it is too expensive to commission a bespoke decision support system, are often not interested in how close solutions are to optimal. They are simply interested in how much better the solutions are than those they currently obtain by hand. For example, consider a small organisation that currently solves its delivery scheduling problem by hand. This organisation may find that the cost of commissioning a team of humans to design a heuristic decision support system, would be far greater than the benefit the company would get in terms of better scheduling solutions. However, the cost of purchasing an 'off the shelf' decision support system which can *automatically* design appropriate heuristics, may be lower than the resulting reduction in costs to the organisation. If the solutions are good enough, and they are cheap enough, then it begins to make economic sense for more organisations to take advantage of heuristic search methodologies. Hyper-heuristic research aims to address the needs of organisations interested in *"good-enough soon-enough cheap-enough"* solutions to their optimisation problems [47]. Note that "good enough" often means solutions better than they currently obtain by hand, "soon enough" typically means solutions delivered at least as quick as those obtained by hand, and "cheap enough" usually

means the cost of the system is low enough that its solutions add value to the organisation.

There are two classes of hyper-heuristic, explained in sections 2.2.1 and 2.2.2. One class aims to intelligently choose heuristics from a set of heuristics which have been provided. The other class aims to automatically generate heuristics from a set of components. It is this second class that is the focus of the work presented in this thesis.

2.2.1 Hyper-Heuristics to Choose Heuristics.

In the majority of previous work, the hyper-heuristic is provided with a set of human created heuristics. These are often heuristics taken from the literature, that have been shown to perform well. When using this type of hyper-heuristic approach, the hyper-heuristic is used to choose which heuristic, or sequence of heuristics, to apply, depending on the current problem state.

On a given problem instance, the performance of existing heuristics varies. Therefore, it is difficult to determine which single heuristic will obtain the best result. When using a hyper-heuristic approach, the strengths of the individual heuristics can potentially be automatically combined. However, for such an approach to be worthwhile, the combination should "outperform all the constituent heuristics" [241]. An optimisation system which intelligently chooses heuristics for the problem at hand can be said to operate at a higher level of generality than the individual heuristics. This is because the system can potentially be applied to many different instances of a problem, and maintain its performance.

A possible hyper-heuristic framework is used in [47, 74, 75, 76, 134, 252]. This is shown in figure 2.1, and has a "domain barrier" between the domain specific heuristics and the hyper-heuristic. This points out the difference in the responsibilities in the model, between the hyper-heuristic and the domain specific heuristics. The hyper-heuristic maintains only knowledge of how many heuristics it has to call upon, and the results of the evaluation of the solutions they obtain.

This idea enabled a tabu-search hyper-heuristic to be applied to the two very different domains of nurse scheduling and university course timetabling in [52]. Different sets of local search heuristics were used for each of the two problems, but the hyper-heuristic was left unchanged. The hyper-heuristic maintains a ranking of its low level heuristics based on their performance, and applies the one with the highest rank at each decision point. If a heuristic is applied and does not result in a better solution, it is placed in

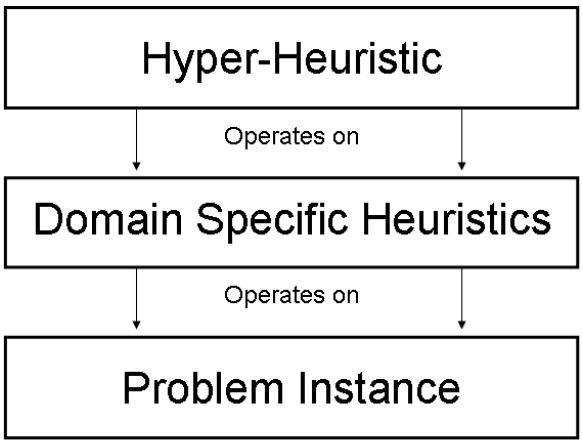

FIGURE 2.1: The relationship between a hyper-heuristic, the domain specific heuristics, and a problem instance [252]

the tabu list and therefore is subsequently not used for a number of iterations. The tabu search hyper-heuristic receives no information about the domain in which it is operating. A domain barrier thus makes it possible to apply a hyper-heuristic to a different problem domain without requiring modification. Similar work on a tabu search hyper-heuristic was presented by Kendall and Mohd Hussin [161]. An improved version of this algorithm is outlined in [160], and tested on a real world timetabling problem from the University Technology MARA.

Another example of a hyper-heuristic that maintains a domain barrier is the choice function hyper-heuristic [74, 252]. At each decision point, the choice function evaluates the domain specific heuristics and chooses the best one [74]. The choice function has three terms. The first is a measure of the recent effectiveness of the heuristic, the second is a measure of the recent effectiveness of pairs of heuristics, and the third measures the time since the heuristic was last called. These three terms represent a trade-off between exploration and exploitation, they make it more likely for good heuristics to be used more frequently, but the third term adds the possibility of diversification. Good results are obtained over the domains of sales summit scheduling, presentation scheduling, and nurse scheduling. Applying the choice function on parallel hardware is investigated in [235].

While the domain barrier has been implemented in the framework above, it is not a defining feature of a hyper-heuristic. A hyper-heuristic may or may not be domain specific.

Often the hyper-heuristic receives little or no information about the problem domain, but it does receive information about the performance of the heuristics being applied. The heuristics must be domain specific because they must operate on instances of the problem domain to obtain a solution.

Many existing metaheuristics have been employed successfully as hyper-heuristics. Both a genetic algorithm and a learning classifier system have been applied to the one-dimensional bin packing problem. The learning classifier system hyper-heuristic [242] selects between eight heuristics every time a piece is to be packed. The percentage of pieces left in four different size ranges is calculated, in addition to the percentage of pieces left in relation to the total number of pieces. The problem state is represented as these two features, which are then compared to a set of rules determining which heuristic will be used to pack the next item. This method learns which heuristics should be used when different features are present. Subsequent work evolves similar rule sets using a genetic algorithm hyper-heuristic [241]. This work shows that it is not only a learning classifier system which can generate rules, and that good results can be obtained by assigning reward to rule sets only when the final outcome of the packing is known. Terashima-marín et al. extend this approach onto the two dimensional stock cutting problem by applying a highly similar learning classifier system [262], and genetic algorithm [264]. A comparison of the two hyper-heuristics for this problem is subsequently given in [265].

Work on a genetic algorithm hyper-heuristic for a trainer scheduling problem is presented in [77, 132, 133, 134]. In this problem, geographically distributed courses are to be scheduled over a period of several weeks, and the courses are to be delivered by a number of geographically distributed trainers [134]. The genetic algorithm chromosome is a sequence of integers that each represent a heuristic. The work is further extended in [134] to incorporate an adaptive length chromosome, so that the length of the sequence of heuristic calls can be modified. Then the results are further improved in [132] by better directing the genetic algorithm in its choice of whether to add or remove genes. A tabu method is added to the genetic algorithm in [133], which means genes are no longer physically added or removed. Instead, they are not used for a number of generations if they do not produce an improvement in the objective function.

An ant algorithm hyper-heuristic is used in [51] for the project presentation scheduling problem. A standard ant algorithm is applied to a search space of heuristics by representing each heuristic as a node in a graph, where an edge between two nodes means one

can be applied after the other. The ants traverse the graph, each producing a solution using the heuristic associated with each node they travel through. An ant lays pheromone on the path it took after a full solution has been constructed, in proportion to the quality of the solution. Thus, the good sequences of heuristics become reinforced. Another ant algorithm hyper-heuristic is presented in [79]. The ant colony algorithm optimises a sequence of five heuristics, each with five parameters. Results are obtained on the two dimensional bin packing problem, making them relevant to the work in this thesis. In their earlier work, Cuesta-Canada et al. employ the term 'hyper-heuristic' to refer to the set of five heuristics, rather than to the ant algorithm which searches the space of these heuristics.

Simulated annealing is employed as a hyper-heuristic in [87] for the shipper rationalisation problem, determining space-efficient sizes for reusable containers. The simulated annealing algorithm is based on the tabu search hyper-heuristic presented in [52], with the difference that once the heuristic is selected by the tabu search, the move it generates is accepted according to the simulated annealing algorithm. A simulated annealing hyper-heuristic is also employed in [13] to automate the design of planograms, which are the two dimensional diagrams used to plan shelf space allocation. Greedy and choice function hyper-heuristics are also investigated in this paper but the simulated annealing had superior performance. Bai et al. present further work inspired by a real world problem in [11], where a collaboration with Tesco informs a study on fresh produce inventory control. Three hyper-heuristics are implemented for this problem, including the tabu search simulated annealing hyper-heuristic previously presented in [87], and in addition to this, metaheuristic and heuristic approaches are also compared. Issues of memory length in a simulated annealing hyper-heuristic are addressed in [12].

The set of domain specific heuristics that the traditional hyper-heuristic chooses between are usually a combination of mutational and hill climbing heuristics. Including heuristics from these two classes means that the search can explore new areas of the search space, and also exploit good areas. Work by Ozcan et al. in [216] explores three different frameworks, which aim to better make use of the strengths of both classes of heuristic. In the first framework, if the hyper-heuristic chooses a mutational heuristic, a hill climbing heuristic is applied immediately afterwards to exploit the diversification before the next heuristic is chosen. In the second framework, only mutational heuristics are available to the hyper-heuristic, and a single hill climbing heuristic is applied immediately after a mutational heuristic is applied, this framework is similar to a memetic algorithm. The third framework

completely separates the sets of mutational and hill climbing heuristics. At each step, the hyper-heuristic chooses and applies a mutational heuristic, and then chooses and applies a hill climbing heuristic.

A hyper-heuristic for timetabling problems is presented in [55], using a tabu search to find a sequence of simple graph colouring heuristics to solve the problem. A genetic algorithm hyper-heuristic is used in [263] for the examination timetabling problem. Using a direct encoding of the problem has been found to be restrictive. For this reason, the genetic algorithm chromosome encodes *instructions and parameters* for guiding a search algorithm, rather than encoding a particular solution. An example of the limitations of a particular direct representation can be found in [240].

Bilgin et al. investigate combinations of seven heuristic selection mechanisms and five move acceptance criteria [23], and the results are also tested on examination timetabling benchmarks. This work shows that many hyper-heuristics in the literature can be viewed as a combination of one heuristic selection mechanism and one acceptance criteria, and that a different combination of these components can be classed as a different hyper-heuristic. Ersoy et al. present similar work using a hyper-heuristic to choose between hill climbers in a memetic algorithm [94]. This, again, shows that a hyper-heuristic can be used to optimise a component of an existing metaheuristic algorithm, not just to choose between fixed algorithms.

Two genetic algorithm hyper-heuristic strategies are analysed for the job shop scheduling problem in [85]. One strategy evolves the choice of which priority rule from twelve to use for conflict resolution at a given iteration. The other evolves the sequence in which the machines are considered by a 'shifting bottleneck' heuristic. This can be called a hyper-heuristic because the space of ordering heuristics is searched for one which minimises the makespan. Storer et al. present work which is highly relevant to hyper-heuristic research, also on the job shop scheduling problem. They state, "Search spaces can also be generated by defining a space of heuristics" [256], which is a fundamental principle of hyper-heuristic research. Specifically, they do this by parameterising existing heuristics and searching the space of parameters. Their subsequent research on how to successfully search such a space is presented in [257].

Hart and Ross have also developed a hyper-heuristic approach for the job shop scheduling problem [136]. They use a genetic algorithm, where each gene represents a combination of scheduling algorithm and heuristic. The scheduling algorithm generates

a set of schedulable operations, and the heuristic determines which among those will be chosen. Zhang and Dietterich use reinforcement learning to learn heuristics for the job shop problem, and their approach is tested on a NASA space shuttle payload processing problem, as well as an artificial problem set. Fang et al. present a genetic algorithm for the job shop scheduling problem, which can be labelled a hyper-heuristic as the genome represents an ordering of the jobs [98]. A subsequent proposed extension to the open shop scheduling problem, in the same paper, involves the genome representing an ordering of both the jobs and the specific tasks that they consist of. They later present the results of this extension in [99]. This paper further expands the hyper-heuristic theme of the work, by encoding into the genome the choice of heuristic that is used to schedule each specific task. In the previous work, a fixed heuristic was used and only the ordering was evolved. The chromosome encodes pairs of values representing which heuristic from eight to apply to which remaining job. The heuristic chooses an operation from the job and places it at the earliest time available in the solution.

Cowling and Chakhlevitch [73] state that the performance of a hyper-heuristic relies very much on the choice of low level heuristics. They address the question of how the set of low level heuristics should be designed. Related to this, is the problem of ensuring that the set of low level heuristics is varied enough to ensure an efficient search, but not so large that it contains heuristics that are not necessary. Further work by the same authors addresses this problem, by introducing learning mechanisms into a hyper-heuristic to avoid using heuristics which do not make valuable contributions [59]. This reduces computational effort, because the under performing heuristics do not then slow the search down.

A case based reasoning hyper-heuristic is presented in [53] for the course timetabling problem, and then again in [56], where it is shown that the hyper-heuristic can operate over both the exam timetabling and the course timetabling domains. This hyper-heuristic works by comparing the current problem to problems encountered before, and applying the same heuristics that have worked well in similar situations. Tabu search is employed in [46] to search for the best combination of two well known graph based heuristics for constructing exam timetabling solutions. The tabu search mechanism is based on that presented in [52] The knowledge gained from this hyper-heuristic is then used to inform two hybrid graph based approaches to the same problem, one which inserts a certain percentage of one heuristic into the heuristic list, and one in which case based reasoning remembers heuristics that have been successfully used on similar partial solutions.

Highly related to the case based reasoning approach is the COMPOSER algorithm [126]. COMPOSER essentially consists of a set of condition-action rules which are learned while working through a representative training set [126]. This algorithm is applied to deep space network scheduling in [124, 125]. The algorithm developed by Fink [105] chooses among three solution methods which apply heuristics in different ways, and is inspired by the PRODIGY architecture that COMPOSER utilises. The solution methods are chosen based on a statistical analysis of their past performance. This is different to case based reasoning, as it does not rely on any prior knowledge of a particular problem domain. This approach focuses on selecting a method to be used throughout the whole search, with no opportunity for switching methods while the search is performed.

Pisinger and Ropke present a hyper-heuristic that can operate over five different variants of the vehicle routing problem [222]. The approach employs adaptive large neighbourhood search, a method which selects a heuristic to destroy part of the solution and a heuristic to rebuild it. Adaptive large neighbourhood search is a paradigm rather than a specific algorithm, it requires an acceptance criteria to be specified as well as the heuristics which drastically modify the solution. The acceptance criteria used in this paper is simulated annealing, but any metaheuristic could be used. The paradigm has similarities to variable neighbourhood search [135], which can also be considered a hyper-heuristic. This is because adaptively changing the neighbourhood of the search can be seen as intelligently selecting an appropriate heuristic.

Wah presents a population based method of learning by examples [273], to create new heuristics. This work led to the development of the TEACHER (TEchniques for the Automated Creation of HEuRistics) learning system, which uses similar genetics based learning to design heuristics [274, 147, 149]. The system is for use in knowledge lean environments, and as such is highly related to the work presented here. An important part of the system is the partitioning of the problem domain into subclasses before creating a heuristic for each. This relates directly to results in this thesis, which show that it is beneficial to evolve a heuristic on a subclass of problems, representative of those on which it will be used in the future. In [274], the system is applied to improve two circuit testing solvers, and more examples of TEACHER applications are given in [148, 247, 261].

The concept of a heuristic to choose heuristics is closely related to the concept of multimeme algorithms [168]. In a memetic algorithm, a meme represents a local search heuristic or some other move operator. The results in [168] show the benefits of using more

than one meme when performing a search. This can also be thought of as a hyper-heuristic employing a number of low level heuristics during a search.

2.2.2 Hyper-Heuristics to Create Heuristics.

Less well studied in the literature is the class of hyper-heuristics which aim to *create* a heuristic from a set of potential components. This is distinct from the class described in section 2.2.1, in which the hyper-heuristic is given a set of complete functioning heuristics, and must then choose between them. The created heuristic may be 'disposable' in the sense that it is created just for one problem instance, and is not intended for use on unseen instances. Alternatively the heuristic may be created for the purpose of re-using it on new unseen instances of a certain problem class.

There are a number of potential advantages of this approach. Every problem instance is different, and obtaining the best possible result for an instance would ideally require a new heuristic or a variation of a previously created heuristic. It would be inefficient for a human analyst to create a new heuristic for every problem instance. Human created heuristics are rarely applicable to only one problem instance, they are usually created for a class of instances, or for all problem instances.

For example, 'best-fit' is a human created heuristic for one dimensional bin packing [162], and performs well on a wide range of bin packing instances. Indeed, it was created as a general heuristic for any bin packing instance. Over a narrower set of instances however, with piece sizes defined over a certain distribution, best-fit can be outperformed by heuristics which are 'tailored' to the distribution of piece sizes [49]. Poli, Woodward and Burke [232] also employ genetic programming to evolve heuristics for bin packing. The structure within which their heuristics operate is based on matching the piece size histogram to the bin gap histogram, and is motivated by the observation that space is wasted when placing a piece in a bin leaves a smaller available space than the size of the smallest piece still to be packed. Their work also differs from that presented in this thesis because they use linear genetic programming, and the problem addressed is offline bin packing (see section 2.3.2) rather than online. However, the motivation for the work is the same, to automate the process of designing heuristics for bin packing.

If the heuristic design process is automated, a computer system could produce a good quality heuristic for an instance in a practical amount of time. This heuristic

could even produce a solution that may be better than that which can be obtained by any current human created heuristic. This is because it would have been created specifically for that instance rather than as a general heuristic. Automating the heuristic design process offers the chance to easily, and flexibly, specify the range of instances that a heuristic will be applicable to, and then obtain that heuristic with minimal human effort spent in the process.

A good heuristic for a given problem instance may be counterintuitive, especially if the instance is complex. Another advantage of automated heuristic design, therefore, is that a heuristic may be created which performs well on the instance, but was unlikely to have been created by a human analyst.

Fukunaga presents an automated heuristic discovery system, for the SAT problem, in [112, 113]. The most successful heuristics for SAT can be expressed as different combinations of a particular set of components. It is stated that human researchers are particularly good at identifying good components, but the process of combining them could benefit from automation [112]. The paper uses an evolutionary algorithm to evolve human competitive heuristics consisting of these components. Some issues and potential objections from [112] are resolved in [113], by implementing the system in 'Lisp' for quicker evaluation, and showing that good heuristics can still be evolved without initialising the population with complex hand coded heuristics and components.

Bader-El-Din and Poli [9] observe that the approach of Fukunaga results in heuristics consisting of other nested heuristics. This results in heuristics which are composites of those in early generations, and which are therefore relatively slow to execute. Bader-El-Din and Poli present a different heuristic generation methodology for SAT, which makes use of traditional crossover and mutation operators to produce heuristics which are more parsimonious, and faster to execute. A grammar is defined, which can express four existing human created heuristics, and allows significant flexibility to create completely new heuristics. Subsequent research by the same team of researchers has shown that a hyper-heuristic system can evolve constructive heuristics for timetabling problems [10]. The evolved heuristics are not intended to be reusable, and the system is presented as an online learning method, which can often obtain better results than other constructive algorithms and hyper-heuristic approaches.

Keller and Poli present a linear genetic programming hyper-heuristic for travelling salesman problems in [159]. At its simplest level, the system evolves sequences of 2-opt

and 3-opt swap heuristics. Then conditional and loop components are added to the set of components, to add complexity to the evolved heuristics. Work presented by Ho and Tay in [140, 260], employs genetic programming to evolve composite dispatching rules for the job shop scheduling problem. The terminals of the genetic programming algorithm are components of previously published dispatching rules, and the functions are the arithmetic operators and an automatically defined function (ADF, see [165]). The evolved dispatching rules are functions, which assign a score to a job based on the state of the problem. When a machine becomes idle, the dispatching rule is evaluated once for each job in the machine's queue, and each result is assigned to the job as its score. The job in the queue with the highest score is the next job to be assigned to the machine. Jakobovic et al. employ the same technique for the parallel machine scheduling problem [151].

Dimopoulos and Zalzala [84] evolve priority dispatching rules for the single machine scheduling problem, to minimise the total tardiness of jobs. The terminal set is based on the human designed 'Montagne' dispatching rule, and contains five elements, representing both global and local job information. The function set consists of the four basic arithmetic operators. While the function and terminal sets are relatively simple, the system evolves heuristics superior to the Montagne, ADD, and SPT heuristics. Genetic programming is also used to evolve priority dispatching rules in [118]. ADFs are not used in this paper, however the function and terminal sets are expanded from that presented in [84, 140]. Human competitive heuristics are produced, even for job shop environments with multiple machines, where a unique dispatching rule is evolved for each.

Recent work by Kumar et al. presents a genetic programming system which evolves heuristics for the biobjective knapsack problem [172]. This is the first work in which heuristics for a multiobjective problem have been automatically generated with a hyper-heuristic. The technique employs many similar ideas to the bin packing methodology presented in this thesis. For example, the terminals used are similar, and the framework which iterates through the pieces, applying the heuristic to each, is similar to the method of iterating through the bins presented in chapter 3. Further work has shown that heuristics can also be generated with genetic programming for a multiobjective minimum spanning tree problem [173].

Oltean [212] presents a linear genetic programming hyper-heuristic, which generates evolutionary algorithms. A standard individual in a linear genetic program represents a series of instructions that manipulate values in a memory array. The memory positions

are referred to as 'registers'. Oltean represents an evolutionary algorithm population as this memory array, with each member of the population stored in one register. The genetic operators are the instructions that operate on the memory array. Tavares et al. [259] also present a methodology for evolving an evolutionary algorithm. They specify the main components of a generic evolutionary algorithm, including initialisation, selection, and the use of genetic operators. Tavares et al. explain how each of these steps, can be evolved individually by a genetic programming hyper-heuristic. They demonstrate the approach through an example of evolving an effective mapping function of genotype to phenotype, for a function optimisation problem.

Pappa and Freitas employ a grammar based genetic programming system to evolve rule induction algorithms for classification [217]. They show that classification algorithms can be automatically specialised to one of two problem classes, a conclusion which is similar to some of the conclusions of this thesis. Their paper states that automatic generation of algorithms may result in better algorithms than have currently been designed by hand, which is a common motivation for hyper-heuristic research. They also assert that such an approach can be cheaper than manual algorithm design. As explained previously in this section, this reduction in cost is another reason why hyper-heuristic research is potentially beneficial to smaller organisations.

The literature also shows a link between memetic algorithms and hyper-heuristics. Krasnogor states a definition of a true memetic algorithm as "evolutionary algorithms where not only the solutions to the problem at hand are evolved but where local searchers that can further improve the quality of those solutions are coevolved, e.g., by GP, alongside them" [168]. Further work by Krasnogor and Gustafson has shown that this is possible, with self generation of memes. For protein structure prediction, a grammar is defined in [169, 170], which expresses groups of memes, each of which performs a local search at a given point in the genetic algorithm. All aspects of the memes are evolved, and the result is a memetic algorithm which is evolved with genetic programming. This provides the possibility of automatically tuning the metaheuristic (in this case a memetic algorithm) to any problem instance [170], and replacing time consuming manual algorithm tuning.

2.3 One Dimensional Packing

This section provides a literature review for one dimensional packing problems. This problem domain is reviewed here because it is one of the domains used to test our hyper-heuristic genetic programming technique. The two and three dimensional packing domains, described in sections 2.4 and 2.5, are also used to test our hyper-heuristic.

2.3.1 Problem Definitions

The Knapsack Problem

The one dimensional 0-1 knapsack problem consists of a set of pieces, a subset of which must be packed into one knapsack with capacity c. Each piece j has a weight w_j and a value v_j. The objective is to maximise the value of the pieces chosen to be packed into the knapsack, without the total weight exceeding the capacity of the knapsack. A mathematical formulation of the 0-1 knapsack problem is shown in equation 2.1, taken from [199].

$$\begin{aligned} \text{Maximise} \quad & \sum_{j=1}^{n} v_j x_j \\ \text{Subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j \in \{0,1\}, \qquad\qquad j \in N = \{1,\ldots,n\}, \end{aligned} \tag{2.1}$$

Where x_j is a binary variable indicating whether piece j is selected to be packed into the knapsack.

The Bin Packing Problem

The classical one dimensional bin packing problem consists of a set of pieces, which must be packed into as few bins as possible. Each piece j has a weight w_j, and each bin has capacity c. The objective is to minimise the number of bins used, where each piece is assigned to one bin only, and the weight of the pieces in each bin does not exceed c. A mathematical

formulation of the bin packing problem is shown in equation 2.2, taken from [199].

$$\begin{aligned}
\text{Minimise} \quad & \sum_{i=1}^{n} y_i & & \\
\text{Subject to} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, & & i \in N = \{1, \ldots, n\}, \\
& \sum_{i=1}^{n} x_{ij} = 1, & & j \in N, \\
& y_i \in \{0, 1\}, & & i \in N, \\
& x_{ij} \in \{0, 1\}, & & i \in N, j \in N, \qquad (2.2)
\end{aligned}$$

Where y_i is a binary variable indicating whether bin i contains pieces, x_{ij} indicates whether piece j is packed into bin i, and n is the number of available bins (and also the number of pieces as we know we can pack n pieces into n bins).

A practical application of the one dimensional bin packing problem is cutting lengths of stock material that has fixed width, such as pipes for plumbing applications, or metal beams [251]. A set of orders for different lengths must be fulfilled by cutting the stock lengths into smaller pieces while minimising the wasted material. Also, the problem of scheduling television advertisements of different lengths into a series of commercial breaks can be represented as a one dimensional bin packing problem [1]. Two practical applications of the knapsack problem are presented in the areas of cryptography [204] and capital budgeting [194].

2.3.2 Previous Work

For the reader looking for an extensive explanation of knapsack problems, Martello and Toth's book [199] is recommended. This goes into detail on a range of knapsack problems including the 0-1 variant which is used in this thesis. There are also good surveys of knapsack problems in [219, 223]. The earliest formalisation of the one dimensional bin packing problem, along with certain production scheduling and other industrial optimisation problems, is that given by Kantorovich [157]. The literature on so called cutting and packing problems is vast, and as such many problems which are otherwise identical are named differently. It is for this reason that Dyckhoff, and later Wäscher et al, have published typologies which attempt to introduce a consistent naming scheme to organise the research performed in this area [90, 276].

TABLE 2.1: A description of existing constructive heuristics for one dimensional bin packing

Heuristic	Description
Best Fit (BF) [236]	Best-fit constructs a solution by putting each piece into the bin which has the least space remaining. The process is repeated until all pieces have been allocated to a bin. No other online algorithm is better in both the average and worst case [162]
Worst Fit (WF) [69]	Puts the piece in the emptiest bin that has room for it. Opens a new bin if the piece does not fit in any existing bin
Almost Worst Fit (AWF) [69]	Puts the piece in the second emptiest bin if that bin has room for it. Opens a new bin if the piece does not fit in any open bin
Next Fit (NF) [153]	Puts the piece in the highest indexed bin and opens a new bin if there is not enough room for it
First Fit (FF) [153]	Puts the piece in the lowest indexed bin that has room for it and opens a new bin if it does not fit in any open bin. An average case analysis of first fit is given in [70]

A number of examples of heuristics used for the bin packing problem are described in table 2.1. They all take a list of pieces as input and pack each piece in turn. While they are all simple, the heuristics have very different behaviour. Figure 2.2 shows where each of the heuristics from table 2.1 would pack a piece of size 60 into the given partial solution. Which heuristic is best depends on the instance,

A bin packing problem is referred to as 'online' if looking ahead is not permitted, the pieces must be packed one at a time, and the pieces cannot be moved once they are allocated a place [35]. The 'offline' version of the bin packing problem occurs when all of the pieces are known before the packing starts. The heuristics listed in table 2.1 are applicable to the online problem because they operate on one piece at a time. These heuristics are often combined with a sorting algorithm which arranges the pieces in non-increasing order before the packing begins. For example, 'first-fit-decreasing' is first-fit applied to a problem after the pieces have been sorted. First-fit-decreasing is therefore an offline heuristic because the pieces must be known for the sorting to occur.

Johnson et al. show that the first-fit-decreasing algorithm has a worst case performance of $(11/9)opt + 4$, where opt is the optimal number of bins. The constant in that expression has subsequently been refined down from four to one in [283]. The performance of this classic algorithm deteriorates when the optimal solution requires the bins to be filled nearly to capacity [127]. Gupta and Ho suggest a heuristic algorithm to address this situation, called 'minimal bin slack' (MBS) [127]. The MBS algorithm attempts to find groups

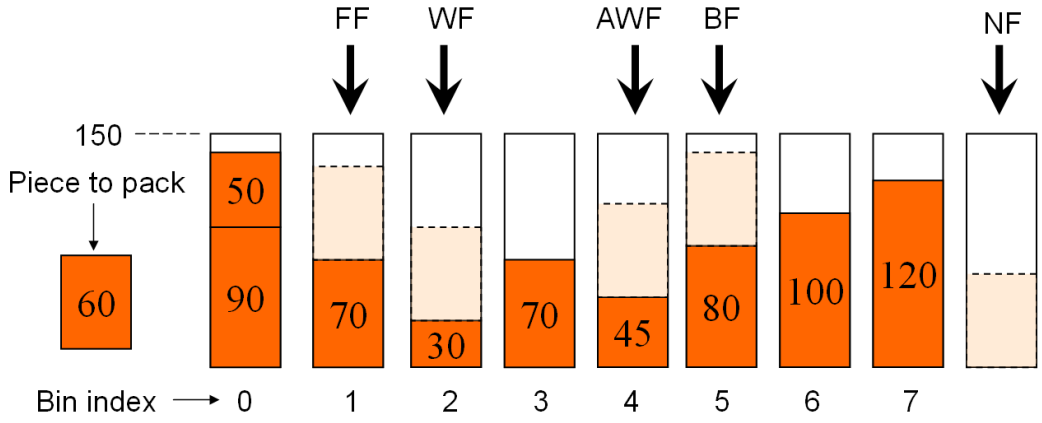

FIGURE 2.2: A diagram showing where the heuristics from table 2.1 would place a piece of size 60. Next-fit only has access to the highest indexed bin and an empty bin, and it chooses the empty bin in this case because the piece does not fit into the highest indexed bin

of items that fit well into a bin. It packs one bin at a time and then moves on to the next bin. For each bin all subsets of items are searched and the subset resulting in the least wasted space is chosen. This algorithm is also guaranteed to find a two bin solution, if one exists. Fleszar and Hindi use a modified MBS heuristic to construct a good initial solution for a variable neighbourhood search. This hybrid algorithm produces better results on the investigated problems than the original MBS heuristic [108].

Two algorithms for one dimensional bin packing are presented in [282]. The first is an online algorithm, with better worst case bounds than first-fit. The second is an offline algorithm, which has better worst case bounds than first-fit-decreasing. Further theoretical work on the performance bounds of algorithms for one dimensional bin packing can be found in [68, 237, 249]

The algorithm presented by Djang and Finch in 1998 first pre-orders the pieces in non-increasing order of size. It then fills a bin to at least one third full by taking the pieces one at a time in order. Then the algorithm searches for a combination of either one, two or three pieces that fill the bin exactly. If no combination is found, then a piece combination that fills the bin to one less than its capacity is searched for, and then two less than the capacity, and so on [242].

Exact methods have been studied for one dimensional packing problems, modelling the problem either as linear or integer programs, and the integer programs are usually solved

by first relaxing the integer constraints to obtain a lower bound. The linear programming approach can be traced back to the work of Gilmore and Gomory [120, 121], and more recent applications of this technique are given in [2, 89]. As well as discussing such linear programming methods, Haessler and Sweeny [131] discuss two more methods. Sequential heuristic methods employ patterns repeatedly, and good patterns must be chosen early to obtain good solutions [130]. The advantages and disadvantages are nicely summarised in [131]. The two approaches can also be combined to form a hybrid approach [131].

Large knapsack and bin packing problems can be represented as integer programs with a large number of variables. Holthaus studies solution methods to large one dimensional bin packing problems with more than one bin size, using a decomposition approach, which first reduces the size of the problem before using the solution to help find a good solution to the original problem [142]. The same problem but with supply bounds is investigated in [19], also formulating the problem as an integer program. The approach combines a cutting plane algorithm with the technique of column generation. Two branch and price algorithms are compared on the standard one dimensional bin packing problem by Vance in [268, 270].

For the one dimensional bin packing problem, Scholl et al. present an exact procedure called 'BISON' [246], based on the well known MTP procedure [199]. In order to find strong upper bounds for the beginning of this procedure, a tabu search heuristic is applied to a modified version of the bin packing problem, where the maximum height of a fixed number of bins is minimised. The tabu search is iterative, and the neighbourhood moves involve swapping two pieces, or moving one piece.

Metaheuristics have also been applied to the one dimensional bin packing problem. For example, a simulated annealing algorithm is used for bin packing in [234], and for a modified version of the bin packing problem in [156, 37]. The number of bins is fixed and the objective is to distribute the pieces so that the height of the fullest bin is minimised. A genetic algorithm has been applied to the one dimensional bin packing problem in [97]. The work is extended in [96] by adding a local search step to the genetic algorithm, based on the concept of dominance of one bin over another. The dominance concept was introduced in [200], along with a reduction algorithm for solving the one dimensional bin packing problem.

Genetic algorithms have also been used to obtain solutions for variations of the knapsack and bin packing problems. For example, Chu and Beasley address the multidimensional knapsack problem with a genetic algorithm in [65]. Also, Raidl and Kodydek use a genetic algorithm for the multiple container packing problem, which combines elements

of both the knapsack and bin packing problems. Chu and Beasley also present a genetic algorithm for the related generalised assignment problem in [64]. Grammatical evolution is employed in [215] to evolve sentences which represent solutions to the multidimensional knapsack problem. Attribute grammars are added to ensure that the constraints of the problem are adhered to.

The technique of weight annealing assigns weights to areas of the fitness landscape, to attract or repel the search process of a greedy heuristic. This technique is used for the bin packing problem in [192], where a weight is assigned to each bin, and swap heuristics are employed to modify the solution. Alvim et al. [6] present a hybrid algorithm involving reductions, constructive heuristics, load balancing, and tabu search. The standard multiprocessor scheduling problem can be stated as a one dimensional bin packing problem, and so heuristics for use in that problem are relevant here. Fujita and Yamashita review the heuristics for use in multiprocessor scheduling in [111], and a tabu search metaheuristic technique is presented in [145].

The heuristics and metaheuristics explained in this section operate directly on a search space of candidate solutions. For example, the genetic algorithm in [97] maintains a population of full assignments of pieces to bins. This is in contrast to a hyper-heuristic approach, where the hyper-heuristic would operate on a search space of one dimensional bin packing heuristics. Examples of such hyper-heuristic approaches can be found in [241, 242], where a learning classifier system and a genetic algorithm operate on a set of domain specific heuristics, which in turn operate on the problem space.

2.4 Two Dimensional Packing

2.4.1 Problem Definitions

The bin packing problem can be defined in two dimensions, where the bin size is defined by a width W and a height H. Each piece $j \in N = \{1, \ldots, n\}$ is defined by a width w_j and a height h_j. The objective is to minimise the number of bins needed to accommodate all of the items, where each item is assigned to one bin only. The pieces must not overlap each other or exceed the boundaries of the bin into which they are placed. In the two dimensional knapsack problem, each piece also has a value v_j, and the objective is to maximise the value of the subset of pieces chosen to be placed in the knapsack. Two dimensional packing problems are also referred to as 'cutting problems' because real world instances of such

problems can involve cutting shapes from sheets of material, while minimising waste.

In this thesis, the genetic programming hyper-heuristic technique is also applied to the two dimensional strip packing problem. This is another cutting problem, analogous to the problem of efficiently cutting shapes from a roll of material. The formulation of the problem involves a sheet of material of fixed width W and infinite height. Each piece $j \in N = \{1, \ldots, n\}$ is defined by a width w_j and a height h_j. When the packing is complete, a horizontal guillotine cut is made across the sheet's width, at a height equal to the upper edge of the highest piece. The objective in this type of problem is to minimise the height of the portion of the sheet that is cut.

Two dimensional packing problems can be separated into many classifications, more than are dealt with in this thesis. For example, in [104, 71, 180, 203, 269], all the pieces to be packed are squares. Typologies for cutting and packing problems are defined by Dyckhoff [90], and later Wäscher et al [276]. The more recent Wäscher et al. typology separates two dimensional packing problems with criteria such as the homogeneity of the pieces, and the number of larger shapes into which the pieces must be packed. Lodi et al. [188] survey the work on two dimensional strip packing and bin packing, including exact, heuristic and metaheuristic approaches. They also include mathematical models of the problems, and lower bounds for their solutions. Dowsland and Dowsland also present a good introductory survey of two and three dimensional problems that occur in the real world in [88]. The literature on two dimensional packing literature is too large to fully review in this section. However, the interested reader is referred to these surveys for an overview of the domain.

In real world applications of two dimensional packing problems, constraints often have to be imposed on the placement of pieces due to properties of the material in use, or properties of the machines that cut the material. One such constraint is that the pieces must not be rotated before they are put into the solution. Another common constraint is the so called 'guillotine' constraint, which means that each cut must be orthogonal from one side of the sheet to the other, creating two new sheets either side of the cut. Subsequent cuts must be made in the same way across the new sheets.

A real world example of the two dimensional packing problem is the crepe-rubber cutting problem, described in [245], in which the pieces cannot be rotated, and the guillotine constraint is applied. Vasko et al. describe a cutting problem from the steel industry where the guillotine constraint is applied, but the pieces can be rotated [271]. The problem of

arranging the layout of newspaper pages is modelled as a two dimensional packing problem in [174]. In this application, the guillotine constraint is not applicable. Also, the orientation of pieces must be fixed because the adverts must be the displayed correctly on the page. Biro and Boros show that a resource constrained scheduling problem can be represented as a strip packing problem [24], and Lee and Sewell model a problem of producing furniture for the boat industry as a strip packing problem, where shapes are cut from rolls of vinyl [178]. The two dimensional packing problem is also related to the domains of space allocation in buildings [43, 44, 45, 250] and shelf space allocation in retail environments [280, 281].

2.4.2 Exact Methods

Exact methods have been shown to work well for small instances. For example, Eisemann [91] presents an early linear programming formulation of a two dimensional stock cutting problem in 1957. Gilmore and Gomory [120] improve on previous linear programming approaches four years later, but the results are still obtained on small instances. A different approach is presented by Dyckhoff in [89]. Tree search procedures have been employed to produce optimal solutions for the 2D guillotine stock cutting problem [63] and two dimensional non-guillotine stock cutting problem [17]. The method presented in [63] has since been improved in [138] and [80].

A branch and bound algorithm is also used in [201] to obtain an exact solution to the problem. Another exact algorithm is presented in [128], which limits the explored tree branches with a bound derived from an integer programming relaxation. Also, an exact algorithm for problems with the guillotine constraint is given in [62], and recently in [67]. Work on exact solutions to general n-dimensional packing problems is presented in [102, 103].

There are many more papers in the literature regarding exact solutions to the two dimensional packing problem, and they are not all reviewed here due to the focus on heuristic methods in this thesis. It is recognised that these methods cannot provide good results for large instances. For example, the largest instance used in both [138] and [80] is 60 pieces. For exact methods, the computational effort required to obtain a solution is exponential in the problem size, and so heuristic methods are more widely used for larger instances.

2.4.3 Heuristic Methods

Heuristic methods cannot guarantee an optimal solution, but can produce good quality results for much larger problem instances. They are often easy for a non-expert to understand, and so are more likely to be applied to real-world problems over other methods. An early heuristic approach to the two dimensional bin packing problem is presented in [21]. The algorithm starts with an initial solution and is based on an iterative process which repeatedly discards the sheet with the most waste so those pieces can be used to improve the utilisation of the other sheets.

Baker et al. [14] define a class of packing algorithms (BL) which maintain bottom left stability during the construction of the solution, meaning that all the pieces cannot be moved any further down or left from where they are positioned. The heuristic presented in [14] has come to be named 'bottom-left-fill' (BLF) because it places each piece in turn into the lowest available position, including any 'holes' in the solution, and then left justifying it. While this heuristic is intuitively simple, implementations are often not efficient because of the difficulty in analysing the holes in the solution for the points that a piece can be placed [60]. Chazelle presents an optimal method for determining the ordered list of points that a piece can be put into, using a 'spring' representation to analyse the structure of the holes [60].

These heuristics take, as input, a list of pieces, and the results rely heavily on the pieces being in a 'good' order [14]. Theoretical work presented by Brown et al. [35] shows the lower bounds for online algorithms both for pre-ordered piece lists by decreasing height and width, and non pre-ordered lists. Results in [143] have shown that pre-ordering the pieces by decreasing width or decreasing height before applying BL or BLF results in performance increases of between 5% and 10%.

Recently, a 'best-fit' style heuristic was presented in [38]. Preordering the pieces for this heuristic makes no difference, the next piece to pack is selected by the heuristic depending on the state of the problem. In this algorithm, the lowest available space on the sheet is selected, and the piece which best fits into that space is placed there. This algorithm is shown to produce better results than previously published heuristic algorithms on benchmark instances [38].

Lodi et al. present four heuristics for two dimensional packing, one for each of the combinations of fixed and 90° piece rotations, and with and without the guillotine

constraint. They also present a more general tabu search framework for two dimensional packing, where the subordinate heuristic can be changed depending on the problem at hand [189].

Work in [179] adds an interactive element to the software package, allowing users to change the solution found by a heuristic. This work draws attention to the assertion that making heuristic software more interactive may encourage greater adoption of heuristic methods as decision support systems in real-world settings.

Zhang et al [284] use a recursive algorithm, running in $O(n^3)$ time, to create good strip packing solutions, based on the 'divide and conquer' principle. Finally, two heuristics for the strip packing problem with a sequencing constraint are presented by Rinaldi and Franz [238], based on a mixed integer linear programming formulation of the problem.

2.4.4 Metaheuristic Methods

Metaheuristics have been successfully employed to evolve a good ordering of pieces for a simple heuristic to pack. For example, Jakobs [152] uses a genetic algorithm to evolve a sequence of pieces for a simpler variant of the BL heuristic. This variant packs each piece by initially placing it in the top right of the sheet and repeating the cycle of moving it down as far as it will go, and then left as far as it will go. Liu and Teng [187] proposed a simple BL heuristic to use with a genetic algorithm that evolves the order of pieces. Their heuristic moves the piece down and to the left, but as soon as the piece can move down it is allowed to do so. However, the BL approach with a metaheuristic to evolve the piece order is somewhat limited, because for certain instances there is no sequence that can be given to the BL heuristic that results in the optimal solution [14, 20].

Ramesh Babu and Ramesh Babu [233] use a genetic algorithm in a similar way to Jakobs, to evolve an order of pieces, but use a slightly different heuristic to pack the pieces, and different genetic algorithm parameters, improving on Jakobs' results. The chromosome allows for multiple sheets to be used for the packing, however the comparison made with Jakobs' results uses only one sheet.

Kröger [171] and Valenzuela and Wang [267] employ a genetic algorithm for the guillotine variant of the problem. They use a linear representation of a slicing tree as the chromosome. The slicing tree determines the order that the guillotine cuts are made and between which pieces (a good explanation and diagram can be found in [171]). The slicing

trees bear a similarity with the genetic programming trees in this paper, which represent heuristics. The slicing trees are not heuristics however, they only have relevance to the instance they are applied to, while a heuristic dynamically takes into account the piece sizes of an instance before making a judgement on where to place a piece. If the pattern of cuts dictated by the slicing tree were to be applied to a new instance, the pattern does not consider any properties of the new pieces. For example, if the slicing tree defines a cut between piece one and piece nine, then this cut will blindly be made in the new instance even if these pieces now have completely different sizes. A heuristic would consider the piece sizes and the spaces available before making a decision.

Hopper and Turton [143] compare the performance of several metaheuristic approaches for evolving the piece order. Each is tested with both the BL constructive algorithm of Jakobs [152], and the BLF algorithm of [14]. The metaheuristics used are simulated annealing, a genetic algorithm, naive evolution, hill climbing, and random search. They are evaluated on benchmark instances, and the results show that better results are obtained when the algorithms are combined with the BLF decoder.

A genetic algorithm for a two dimensional knapsack problem is presented in [129], and different versions of genetic algorithms are also employed in [146], for three types of two dimensional packing problems, including strip packing and bin packing. A heuristic approach using a local search on a sequence pair representation is presented in [221], with a simulated annealing acceptance criteria. They test these heuristics both on the classical instances and on new instances which they define. The setting of good parameters when using a simulated annealing algorithm for packing is investigated in [86], and simulated annealing is employed in [39] to evolve a piece order for the best-fit heuristic presented previously in [38].

Other approaches start with a solution and iteratively improve it, rather than heuristically constructing a solution after optimising the piece order. Lai and Chan [175], Faina [95], and Dagli and Hajakbari [81] implement simulated annealing algorithms. Also, Bortfeldt's genetic algorithm [29] operates directly on the representations of strip packing solutions.

A reactive greedy randomised adaptive search procedure (reactive GRASP) is presented in [4] for the two-dimensional strip packing problem without rotation. The method involves a constructive phase and a subsequent iterative improvement phase. To obtain the final overall algorithm, four parameters were chosen using the results from a computational

study. First, one of four methods of selecting the next piece to pack is chosen. Second, a method of randomising the piece selection is chosen from a choice of four. Third, there are five options for choosing a parameter δ, which is used in the randomisation method, and finally there are four choices for the iterative improvement algorithm after the construction phase is complete. It is a complex algorithm with many parameters, which are chosen by hand to enable the algorithm to obtain some of the best results in the literature.

Belov et al. have obtained arguably the best results in the literature for this problem [20]. Their 'SVC' algorithm is based on an iterative process, repeatedly applying one constructive heuristic, 'SubKP', to the problem, each time updating certain parameters that guide its packing. The results obtained are very similar to those obtained by the GRASP method. They obtain the same overall result on the 'C', 'N' and 'T' instances of Hopper and Turton, but SVC obtains a slightly better result on ten instance sets from Berkey and Wang, and Martello and Vigo. Together, SVC(SubKP) and the reactive GRASP method in [4] represent the state of the art in the literature, and SVC(SubKP) seems to work better for larger instances [20].

2.5 Three Dimensional Packing

2.5.1 Problem Definitions

In a three dimensional bin packing problem, each bin is defined by a width W, a height H, and a depth D. Each piece $j \in N = \{1, \ldots, n\}$ is defined by a width w_j, a height h_j, and a depth d_j. The objective is to minimise the number of bins needed to accommodate all of the items, where each item is assigned to one bin only. The pieces must not overlap each other or exceed the boundaries of their bin. In the three dimensional knapsack problem, there is one bin available, with dimensions W, H, and D. Each piece j also has a value v_j in addition to w_j, h_j, and d_j. The objective is to maximise the value of the pieces chosen to be packed into the one bin.

There are numerous other three dimensional packing problems. For example, one special case involves packing only cubes [203, 209]. In this thesis, heuristics are evolved for instances with differing dimensions only, but while we are aware of the wide variety of other problems that have been introduced in the literature, there are too many to provide a full review here. The typologies of Dyckhoff [90], and more recently Wäscher et al [276] provide classifications for these problems, in response to the vast literature in which references to

identical problems often use different names. Dowsland and Dowsland provides a good survey of the different types of real world two and three dimensional problems [88].

Three dimensional packing problems are abstractions of many real world problems, and it is possible to model real world problems as classical three dimensional packing problems with additional constraints. A prominent example is the problem of loading a container ship, where the goal is to maximise the volume utilisation of a single ship, or to minimise the number of ships that are needed to hold a given set of containers. Beasley and Hoare [18] study a real world shelf space allocation problem, which is essentially a three dimensional packing problem with added constraints. Other real world examples are cutting foam rubber in arm-chair production, designing packaging [190], and a packing algorithm being developed for a French biscuit factory [36].

However the constraints faced in these real world situations are often not reflected in the problem instances studied in the literature. For example, Bischoff and Ratcliff argue that the common metric of volume utilisation of the container is insufficient for real world problems where many other constraints must be taken into account [25]. They address two of these constraints with two algorithms. One algorithm creates stable packings, and one algorithm creates convenient packings, assuming that the pieces are to be unloaded at more than one stop. The pieces' load bearing strength is addressed by an algorithm by Bischoff in [26], which ensures that heavier pieces are not placed on top of pieces that cannot support their weight.

2.5.2 Exact Methods

There are relatively few examples of exact methods being used for three dimensional packing problems, when compared to the literature for one or two dimensional problems. Exact methods are perhaps less appropriate due to the greater number of arrangements per piece, and therefore the greater number of calculations that must be done. For example, each piece can be oriented in six ways in a three dimensional instance, compared to only one way in a one dimensional instance, and there are many more potential locations for a piece within a bin. There are often more location and orientation combinations than can be evaluated in a reasonable amount of time for even small three dimensional instances.

An exact method using an integer programming representation is presented in [150]. Martello et al. base a branch and bound algorithm on an exact method for filling one

bin. Results can be obtained in reasonable time with this algorithm for instances of up to 90 pieces [197]. However, it is shown in [83] that not all orthogonal packings can be generated with this algorithm, therefore it may miss an optimal packing. Also, better lower bounds are found by Boschetti in [33], who concludes that this means more instances can be solved to optimality by the exact algorithm in [197], and that it would require less computing time. Changes to the algorithm are presented in 2007 by Martello et al, which ensure that all packings are considered [198]. In this paper, Martello et al. draw attention to the fact that in the real world a robot arm can be used to pack a container. This will constrain the potential locations of a piece because the arm must be able to place the piece without collisions with other pieces. The idea of an 'envelope' to represent the partial solution is explained, a piece cannot be placed in a position where an existing piece would be in front of, right of, or above it. The robot packing constraint provides part of the inspiration for the 3D packing representation in this thesis, as does the idea of an 'envelope' behind which no piece can be placed.

2.5.3 Heuristic and Metaheuristic Methods

Pisinger and Egeblad use a heuristic approach using 'sequence triples' for the three dimensional representation [221]. This approach is based on the sequence pair representation for two dimensional problems, mentioned in section 2.4.4, and the same simulated annealing strategy is applied. Eley presents an approach which uses a greedy heuristic to create blocks of homogeneous identically oriented items, and a tree search afterwards to improve upon the total arrangement by changing the order in which the piece types are packed [92]. Another approach employing a greedy heuristic and tree search is presented in [185], for both the bin packing and the knapsack problem. A feature of the paper is a system for assigning 'blame' to problem pieces, meaning that they will be packed earlier in the sequence on the next iteration. This is similar to the hyper-heuristic approach to exam timetabling of Burke and Newall [40], which adapts the heuristic order at each iteration. Both of these papers draw upon 'squeaky wheel' optimisation, presented by Joslin and Clements [154].

Li and Cheng present several algorithms for the variant of the problem with no rotations allowed [182], including an algorithm for use if the underside of each piece is a square. The same authors also developed an algorithm two years later with a better asymptotic performance bound [183]. Miyazawa and Wakabayashi compare their algorithm

to this, showing that it has an asymptotic performance bound which is better still, answering a question raised by Li and Cheng on whether an algorithm exists with a better performance bound than theirs [207]. In 1999, Miyazawa and Wakabayashi present four algorithms for four similar problems [208]. They relax one of the packing constraints by allowing rotations around the z axis. The algorithms have slightly better asymptotic performance bounds to the previous work in the literature. The good performance bounds come from careful analysis of the structure of the problems in question. Whereas Miyazawa and Wakabayashi perform this analysis manually, one of the aims of hyper-heuristic research is to create systems which can automate this process.

Ngoi et al. describe an intuitive heuristic procedure which constructs a solution by placing a piece in the position which results in the least wasted space around it [211]. A similar spatial representation technique is presented in [66], also incorporating a user interface and the ability to specify certain locations for items before the packing begins. Lim et al. use a multi-faced buildup method, allowing every surface in the container to be a 'floor' to locate boxes [184]. An algorithm based on the concept of 'caving degree' is introduced in [144]. Caving degree is a measure of how close a box is to those already packed, and the packing with the largest caving degree is chosen.

Pisinger uses a wall building approach to obtain good results on a container loading problem [220], and also explains two other approaches to the problem, namely the stack building approach and the cuboid arrangement approach. In general, the wall building approach separates the container into distinct layers along its depth, and then each layer is filled with pieces. This approach was introduced by George and Robinson [119] who, in contrast to later wall building algorithms, allow pieces to straddle two layers. Moura and Oliveira [210] combine the basic George and Robinson algorithm with a GRASP strategy.

Pisinger's approach uses a tree search to determine the depths of the layers, "*each layer is then filled as a number of strips, where the strips may be oriented horizontally or vertically.*" [220]. Bischoff and Marriott compare 14 heuristics based on the wall building approach [27]. Another example of a wall building approach is presented in [117], where a number of solutions are created and the user can choose the best one. The hybrid genetic algorithm of [31] creates an initial population with a basic heuristic which forms vertical layers in the container. These layers are then used as the unit of crossover and mutation in the genetic algorithm. In general, genetic algorithms have not been widely studied for three dimensional packing problems, possibly due to the difficulty in creating an effective

representation. However, some examples include, but are not limited to, [72, 158, 186].

The stack building approach involves creating 'towers' of pieces, and arranging them efficiently on the floor of the container. An example of this class of algorithms is shown in [116], where a simple heuristic creates the stacks and a genetic algorithm is applied to arrange the stacks onto the container floor. The cuboid arrangement approach packs groups of similar pieces. An example of this approach is presented in [30], where a group can be made up of one or two types of pieces, and a tabu search metaheuristic is used to improve the solution iteratively. The cuboid arrangement approach is useful when stability is a primary concern, because it produces very stable packings, with similar pieces stacked with each other.

Lodi proposes a tabu search method, in conjunction with a heuristic which fills the container in layers along its height [190]. This heuristic takes into account the need for a good vertical filling, with items of similar height packed into a layer, and also the need for efficiently packed layers, which is essentially a two dimensional packing problem. A tabu search method for container loading is also presented in [32], with an emphasis on the stability constraint, and how to parallelise the algorithm. Faroe et al. implement a guided local search technique for the three dimensional bin packing problem [101], The guided local search penalises certain features of a solution when a local optimum is found. Similar to the methodology presented in this thesis, they apply the technique to two dimensional instances by setting the pieces' depth dimension to a constant.

2.6 Complexity Status

Sections 2.3 to 2.5 have presented the various exact, heuristic, and metaheuristic methods which have been employed for the one, two and three dimensional packing problems. This section briefly summarises the complexity status of the problems addressed in this thesis.

The one dimensional bin packing problem is well known to be NP-Hard [5]. As such, the literature concerning this problem mainly presents heuristics and approximation algorithms. However, the one dimensional knapsack problem is weakly NP-Hard, as it can be solved in pseudo polynomial time using dynamic programming [115]. The two dimensional strip packing problem is NP-Hard, as a one dimensional bin packing problem can be easily converted into a two dimensional strip packing problem [5, 196].

The three dimensional bin packing problem is strongly NP-Hard, as it is a general-

isation of the one dimensional bin packing problem [196]. The three dimensional knapsack problem (container loading problem) is also strongly NP-Hard, and both of these three dimensional problems are considered very difficult to solve in practice [100, 196, 220]. Heuristic and metaheuristic search techniques are most common for these problems. Nevertheless, an exact algorithm for the three dimensional knapsack problem has been presented which can solve instances with up to 90 pieces [196].

2.7 Genetic Programming

Genetic programming is an evolutionary computation method which evolves computer programs [15, 164, 167]. Despite the similar name, the methodology is distinct from a genetic algorithm. In a genetic algorithm, the chromosome represents a candidate solution to a problem, usually as a fixed length string of characters. In a genetic programming run, the chromosome represents a program, or a method to solve a problem, usually as a variable size tree structure. Importantly, the user does not have to specify the size and shape of the final solution that they expect.

A hyper-heuristic is defined as a heuristic which searches a space of heuristics, as opposed to directly searching a space of candidate solutions. Koza states that the search space of genetic programming is a search algorithm over "*the space of all possible computer programs composed of functions and terminals appropriate to the problem domain.*" [164] Intuitively, one can see how genetic programming may lend itself to the class of hyper-heuristic which aims to generate heuristics (methods to solve a problem) from a set of potential components (see section 2.2.2).

Genetic programming became popular due to the work by John Koza in 1992 [164]. A good account of the previous work on evolutionary algorithms, which led to the development of genetic programming can be found in [15]. To paraphrase some of this history, employing a tree structure for the individuals in a genetic programming population was suggested by both Cramer [78] and Koza [163], and some of the first work evolving programs using the 'Lisp' and PROLOG programming languages can be found in [137, 110].

Section 2.7.1 will explain the standard structure of a genetic programming chromosome. Section 2.7.3 will describe five preparatory steps that need to be completed before a genetic programming run can start. Section 2.7.4 will explain the three basic genetic operators, and how they operate on the chromosome structure defined in section 2.7.1. Some

of the theoretical understanding of genetic programming is introduced in section 2.7.5, and some of the achievements of genetic programming to date are presented in section 2.7.6.

2.7.1 Program Structure

In genetic programming, the programs that comprise the population are traditionally represented as tree structures. There are other program structures which can be evolved. For example, linear genetic programming [34, 218, 228] evolves linear sequences of instructions. Graph structures are evolved in parallel distributed genetic programming [225, 224] and cartesian genetic programming [205, 206]. Tree based genetic programming is the technique employed in this thesis. What follows is a short introduction to this methodology.

Each node in the tree returns a value to its parent node. The leaf nodes are usually input variables providing information about the problem state, or numeric constants. The internal nodes have one or more children nodes, and they return a value obtained by performing operations on the values returned by their children nodes. In genetic programming, the trees' internal nodes are referred to as 'functions', and leaf nodes are referred to as 'terminals'.

For example, consider a one dimensional bin packing problem (see section 2.3.1). Figure 2.3 shows a program which calculates the space left at the top of a bin, such as the one shown in figure 2.4, where a piece is put into a bin which is already partially filled. The terminals in figure 2.3 are 'C', 'S', and 'F'. C represents the capacity of the bin. S represents the size of the piece that is put into the bin. F represents the fullness of the bin before the piece of size S is put in. Figure 2.4 shows the values that these terminals correspond to.

The functions in figure 2.3 are the arithmetic operators of '$-$' and '$+$'. The '$+$' function takes two arguments, S and F, sums them, and returns the result to its parent node. Its parent node is the '$-$' function, which also takes two arguments, C and the value returned by the '$+$' node. It subtracts the second argument from the first, and returns the result as the output of the program.

2.7.2 Initialisation

At the start of a genetic programming run, the population must be initialised. A number of methods have been proposed for this process, and in most cases, the trees' nodes are

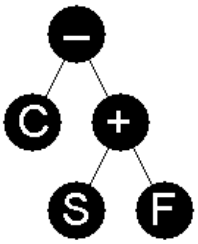

FIGURE 2.3: An example tree structure representing a program that calculates the space left in the bin in figure 2.4

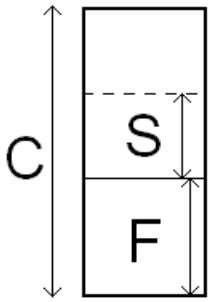

FIGURE 2.4: The features of a one dimensional bin to which the tree in figure 2.3 could be applied to

selected randomly from the manually defined set of functions and terminals. For the three methods explained here, the trees have a predefined maximum depth to limit the size of the initial trees. The 'full' method selects a function for every node until the maximum depth is reached, where a terminal must be selected. The result is a tree with all of the terminal nodes at the same depth. The 'grow' method has no constraints on where the terminals can be placed, resulting in trees with varied structures. The 'ramped half-and-half' method was proposed by Koza [164], and is the initialisation method used in this thesis. In this method, half of the population is initialised with the grow method, and half with the full method. The maximum depth of the trees is also varied, creating a wide range of initial tree structures.

2.7.3 The Five Preparatory Steps

For a basic run of genetic programming, the following five steps must be completed, taken from [167].

Define a Terminal Set

The first step is to "*define the set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program*" [167]. The terminal set is the set of nodes that can be the leaf nodes of the program tree, and as such, they take no arguments. They are the mechanism through which the problem state is made available to the program. They act as input variables, changing

their value as the problem state changes. For example, in the program in figure 2.3, the value of the F terminal will decrease if the bin has a lower fullness, and the value of S will increase if the piece put into the bin is larger. So the program will return a different result in either case.

The example of evolving a program to control a robot to clean a floor is given in [167]. The terminals may be defined as the movements that the robot can make, such as 'turn right', 'turn left', 'move forward', and 'swish the mop'. Other terminals may provide sensory information, such as how far an obstacle is from the robot [167]. Terminals can also be constant values that may be useful to the program. For example, a program calculating the gravitational force between two objects, will need the universal gravitational constant declared as a terminal, which will not change as the objects are changed.

Define a Function Set

The second step is to define "*the set of primitive functions for each branch of the to-be-evolved program.*" [167]. The function set is the set of operations that can be represented by the internal nodes of the tree. For example they can be arithmetic operators, boolean logic operators, or conditional operators. The functions of a genetic programming tree manipulate the values supplied to the program by the terminals, and as such their defining feature is that they take one or more arguments, which can be the values returned by terminal nodes or other function nodes.

Define a Fitness Measure

The third step is to define "*the fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population)*" [167]. The fitness measure (or objective function) assesses how good a program is at the problem at hand, and the result is used to select individuals to be parents of those in the next generation. To return to the cleaning robot example, the fitness measure may be defined as the percentage of the floor successfully cleaned by the robot.

Define the Run Parameters

The fourth step is to define "*certain parameters for controlling the run*" [167]. These include the population size, the maximum program size, and the percentage of the new population

that is to be created from each of the genetic operators.

Define the Termination Criterion

The fifth step is to define "*the termination criterion and method for designating the result of the run*" [167]. For example, the run could stop when a maximum number of generations has been executed, or a time limit has been reached. Alternatively, the run could stop when an individual of a certain quality has been evolved.

2.7.4 Genetic Operators

In a 'canonical' GP implementation, there are three genetic operators which modify the population to create the next generation. They are 'crossover', 'mutation', and 'reproduction'. Originally, the work by Koza in [164] did not include the mutation operator. However, there is now general consensus that mutation is beneficial to the search and should be included. In the experiments in this thesis, a generational evolutionary model is employed, rather than a steady state model. In each generation, each individual in the population is assigned a fitness value in proportion to their performance on the given task. As is the case with other evolutionary algorithms, the fitness values are then used to determine which individuals are to be the parents of those in the next generation. The next generation is created by applying a combination of the genetic operators to create new individuals. This is in contrast to a steady state model, where one genetic operator would be applied once to create a new individual, which would replace one individual in the population and be tested immediately.

For example, fitness proportional selection can be used, in which each individual is selected to be a parent with a probability proportional to their fitness. However, this can mean that one strong individual can dominate the population if its fitness is disproportionately higher than any other individual's fitness. This problem can be addressed in a number of ways, one of which is with a tournament selection mechanism. In tournament selection, two or more individuals are selected from the population, and the individual with the highest fitness becomes a parent.

Crossover

The standard crossover operator requires two parent individuals, and creates two children. A node is chosen at random in each individual, and the subtrees defined by each point are exchanged. This process creates two child individuals, which are added to the next generation. Figure 2.5 shows, graphically, the crossover operator being applied to two individuals. Uniform selection of crossover points often leads to a bias towards simply swapping two leaf nodes, as the average number of children is usually at least two. There is therefore more chance that a leaf node is chosen than an internal node. Koza suggests choosing an internal node in 90% of cases, and a leaf node in 10% of cases, in order to force the crossover of more complex subtrees. This is the approach taken in this thesis.

Mutation

The mutation operator requires one parent individual, and creates one child. A node is chosen at random in the parent, and it is common to force the selection of an internal node 90% of the time. The subtree defined by that node is deleted, and a new subtree is generated in its place, consisting of nodes chosen from the function and terminal set. When only using crossover, a function or terminal can be lost from one generation to the next if no individuals that contain it are chosen to be parents. Mutation provides a mechanism for a lost function or terminal to re-enter the population. In general, mutation can be a way to diversify the search and help avoid local optima. Figure 2.6 shows this process graphically.

Reproduction

The reproduction operator simply copies one parent to create one child. Mutation and crossover can be destructive. They can cause a good individual to be lost from one generation to the next. The reproduction operator provides a mechanism to retain good individuals, so that they can be further modified after the next generation. In this thesis, the evolutionary model is always generational, rather than steady state. The next generation is created in one process using a combination of the genetic operators, and if the reproduction operator is called, it makes an exact copy of one individual and places it in the next generation.

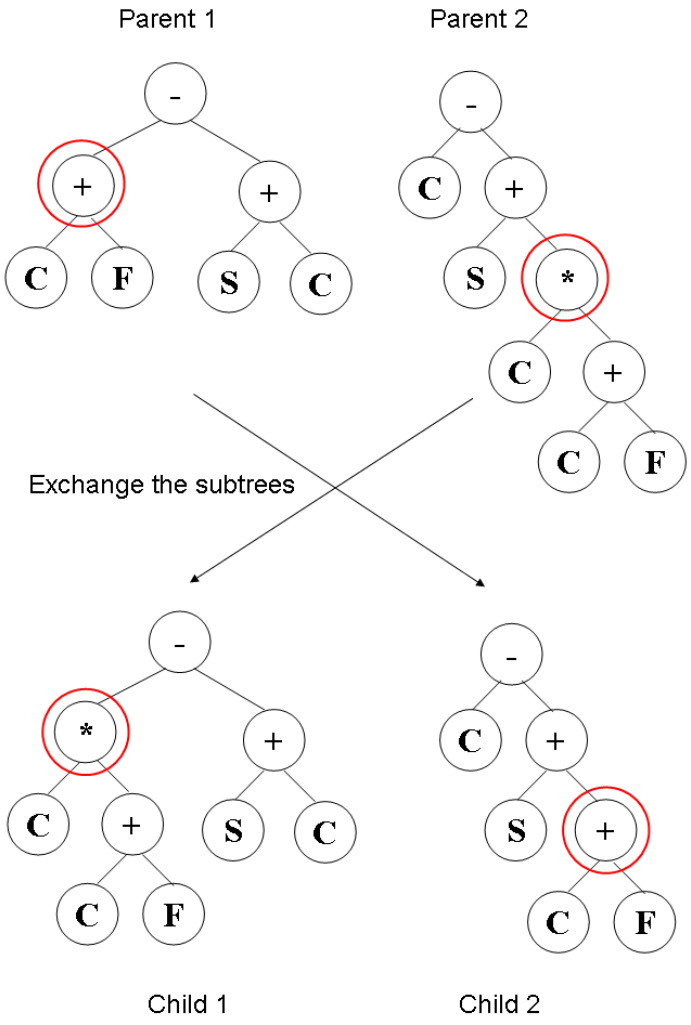

FIGURE 2.5: An example of a genetic programming crossover operation being performed on two parent trees to create two child trees

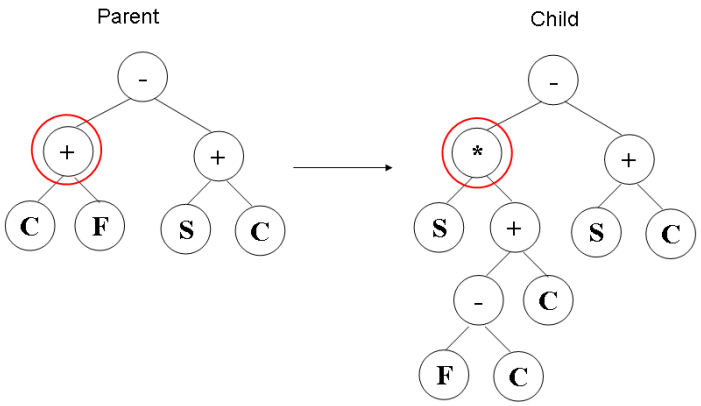

FIGURE 2.6: An example of a genetic programming mutation operation being performed. Only one parent tree is necessary, and one child tree is produced. The subtree rooted at the circled node is deleted, and a new subtree is grown at the same point, by randomly selecting each node

Constraints on the Tree Structure

In standard genetic programming, as the trees can be arbitrarily recombined with the crossover operator, it is important that the functions and terminals satisfy a constraint called 'closure' [164]. This means that any output of a function or terminal can be used as the input to any other function. If this is not the case, then it cannot be guaranteed that the crossover operator will produce syntactically valid trees. However, ensuring closure of the function set is not the only way to ensure that trees produced with crossover will execute. Indeed, sometimes it may actually be necessary to make use of a set of functions which returns differing types of outputs, and numerous methods have been proposed to extend genetic programming in order to handle this. Poli et al. [228] presents a summary of these approaches, including strongly typed genetic programming, and grammar based approaches.

2.7.5 Theoretical Concepts

Banzhaf et al. [15] provide a chapter outlining the biological concepts that underpin the inspiration for genetic programming, and also a chapter on the mathematical basics needed to understand certain genetic programming concepts. Currently, the most developed theoretical explanation of genetic programming is based on Holland's 'schema theory' [141]. Schemata are building blocks which help the individual to solve the problem. In genetic

algorithm theory, they are represented as sub-sections of the chromosome with 'wildcard' characters for genes which are unspecified. It has been less straightforward to adapt this model to genetic programming, because the chromosome is of variable length, and the crossover operators used are more complex. As such, it has been more difficult to agree on a similar definition for schemata [15, 227].

Koza first adapted schema theory to genetic programming in his 1992 book [164]. And more recent work on schema theory has been done by Poli et al. [229, 230, 231]. A good survey of all the work on schema theory, and other theoretical models of genetic programming, can be found in chapter 11 of [228].

2.7.6 Achievements of Genetic Programming

Genetic programming has been applied to many different kinds of problems, and it would be impossible to list all of its achievements in this literature review. This section will give some of the main areas in which genetic programming has been successful.

The 2009 Genetic and Evolutionary Computation Conference will host the sixth annual awards for human competitive results produced by genetic and evolutionary computation. While results achieved by any evolutionary computation technique can be submitted, the results of the previous human competitive awards are a showcase for the ability of genetic programming to create innovative solutions to real world problems. There are eight criteria by which the entries are judged, ranging from infringement on an existing patent, to "holding its own" in competitions with humans. The full details can be found at the competition website:

http://www.genetic-programming.org/hc2005/main.html

Poli et al. cite two prominent examples of work that fulfils many of these criteria. The first is an antenna design evolved by genetic programming [193]. The design had a performance very similar to the design created by the contractor commissioned to design the antenna. The second example is the design of algorithms for quantum computers [254, 255].

Poli et al. [228] also report a number of additional areas where genetic programming has proved to be successful, the reader is referred to [228] for an extensive list of references, but some are provided here: image and signal processing [82, 195, 258], financial trading and economics [61, 155, 266], industrial process control [58, 109, 181], bioinformatics

and medicine [3, 28, 93], hyper-heuristics [49, 114, 159], and entertainment and computer games [8, 177, 272].

2.7.7 Summary

To conclude this section, the literature on genetic programming is vast, and it would be impossible to completely cover in this review. For example, the genetic programming bibliography contains over 5000 references. It is maintained by William Langdon, Steven Gustafson, and John Koza, and can be found at this website:

http://www.cs.bham.ac.uk/∼wbl/biblio/

This section has covered the sections of the genetic programming literature that are relevant to this thesis, and the reader who is interested in learning more about genetic programming is directed to the excellent book by Poli et al. [228], and to these additional references [15, 164, 165, 166].

2.8 Conclusion

The aim of this chapter was to put the contribution of this thesis in context. Previous work has been described for the areas of hyper-heuristics, packing in up to three dimensions, and genetic programming.

Hyper heuristics have been defined as heuristics which search a space of heuristics, as opposed to directly searching a space of solutions. Since the first few publications of hyper-heuristic methods in the 1960s (see [107] for the first), little progress was made to exploit this approach to the development of search methodologies until the 1990s. In the last decade, much more research has been undertaken into hyper-heuristics. It can be reasoned that hyper-heuristic methodology has seen a resurgence of interest recently because of the increase in difficulty and scale of problems in the real-world [252]. While the difficulty has increased, the cost of solutions has increased, because the same methods based on hand crafted solutions are applied to progressively more difficult problems, which takes more time in human analysis and development. Large companies can afford to commission such bespoke systems. However, smaller organisations with optimisation problems do not have access to them, because of the prohibitive cost.

There is a need, therefore, for an automated approach to heuristic generation and selection, which can provide value for money to users who do not have the funding to commission a bespoke system. The hyper-heuristic research presented in this chapter shows the first steps to providing more general, more automated, and therefore cheaper systems.

The one, two and three dimensional bin packing and knapsack problems have been introduced, along with the two dimensional strip packing problem. There has been a large amount of previous work on these problems, especially in one and two dimensions. The previous work has generally been to develop heuristics for one of these problems only. In the vast majority of cases, a human-created heuristic for one problem type cannot be employed for another problem type without changing the heuristics parameters at the very least, because they are designed and built by hand specifically for one problem. This thesis aims to show that a computer can be employed to design and build heuristics for any of the packing problems presented in this chapter, thus automating this potentially expensive process usually done by hand.

The wealth of literature on genetic programming mostly studies concepts beyond the complexity of this system. However, for the reader not familiar with genetic programming, section 2.7 provides a strong starting point to understanding the genetic programming system used in this thesis at least. References are provided to good introductory texts if more information is needed.

In this thesis, we use genetic programming as a hyper-heuristic for a variety of packing problems. The next chapter presents initial work on a hyper-heuristic genetic programming system for one dimensional bin packing.

Chapter 3

Evolving Heuristics for the One Dimensional Bin Packing Problem

3.1 Introduction

In the previous chapter, we presented a review of the relevant literature regarding hyper-heuristics, packing problems, and genetic programming. This chapter will present initial work on automating the design of heuristics for the one dimensional bin packing problem. In subsequent chapters we will expand and improve upon this initial work, showing that it is possible to create human-competitive heuristics with genetic programming for well known packing problems.

The aims of this work are to investigate if it is possible for a genetic programming system to generate heuristics equal to, or better than, human created constructive heuristics. If this is found to be possible then it would validate the hypothesis that genetic programming can be employed as a hyper-heuristic to generate heuristics. Experiments are performed to determine the relative effectiveness of two fitness functions. We also investigate the effects of removing a particular function from the function set.

As we have seen in section 2.7 of the previous chapter, genetic programming evolves a population of tree structures that represent programs. The heuristics we evolve are therefore tree structures, with leaf nodes that represent certain features of the current problem instance state, and with internal nodes that perform operations on those feature values to return one numerical result. At each generation, each heuristic in the population is applied to a set of problem instances, and assigned a fitness based on its performance. The fitness scores of all the heuristics are then used to determine which ones will be selected

as parents for the individuals in the next generation.

The trees we evolve operate within a fixed framework where the tree is applied in turn to the available bins. The tree returns a value for every bin. If the value is zero or less then we move on to the next bin, but if the value is positive we put the piece in the bin. In this way, it is the tree which decides when to stop the search for a suitable bin and put the piece into the current bin. The tree therefore operates as a heuristic. The algorithm then repeats the process for each of the other pieces until all the pieces have been packed.

This is similar to the way first-fit operates, and so we will explain this heuristic in section 3.2. The methodology is explained in section 3.3. Results are displayed in section 3.4, and conclusions are drawn in section 3.5.

3.2 The First-Fit Heuristic

First-fit is a constructive heuristic which packs a set of pieces one at a time, in the order that they are presented. The heuristic iterates through the open bins, and the current piece is placed in the first bin into which it fits. This heuristic is for the online bin packing problem, because it packs pieces one at a time, and a piece cannot be moved once it has been assigned to a bin. The difference between an online problem and an offline problem is explained in section 2.3.2.

Figure 3.1 shows an intermediate stage of a simple packing problem, with four pieces left to pack and some pieces already packed into the partial solution on the right. The current piece (with a size of 70) cannot fit into the first bin or the second bin, because they do not contain enough free space. However, the piece fits into the third bin, so the piece will immediately be committed to that bin, as shown in figure 3.2. The algorithm will then return to the first bin, this time considering the second piece (with a size of 85). It will iterate through the bins again, this time to the fourth bin, and immediately choose to put the piece there, as shown in figure 3.3. This is because the fourth bin is the first with enough free space to accommodate it. The final two pieces are packed in the same way, and the final solution is shown in figure 3.4.

In summary, this heuristic iterates through the bins, and at each bin it makes a final decision on whether or not to place the piece into the bin. We draw our inspiration from first-fit for our evolved heuristics. In general terms, we will evolve the heuristic part of the algorithm, which will choose whether or not to put a piece in the current bin or move

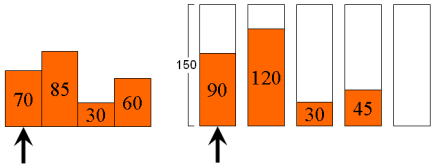

FIGURE 3.1: Pieces remaining to be packed, and a partial solution

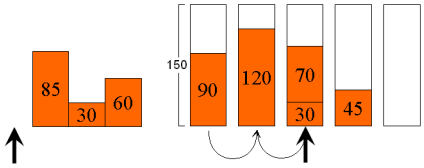

FIGURE 3.2: The first piece is put into the third bin because it is the first bin with enough space to accommodate it

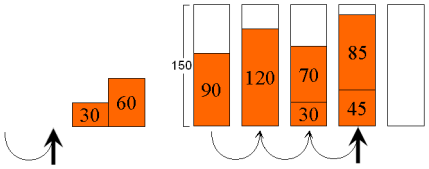

FIGURE 3.3: The second piece is put into the fourth bin

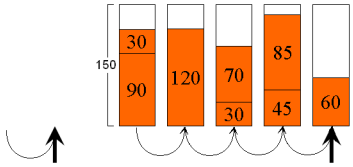

FIGURE 3.4: The final two pieces are put into the first and fifth bins

on to consider the next bin.

3.3 Methodology

In this section, we explain how a heuristic in the population is applied to a problem instance, what the heuristics actually consist of, and how we evolve the heuristics. Section 3.3.1 explains how we apply a heuristic to the problem to obtain a result. Section 3.3.2 explains what the heuristics consist of. In section 3.3.3, we give a detailed example of part of the packing process, to further clarify the process of applying a heuristic. Section 3.3.4 gives examples of heuristics that produce illegal solutions. Finally, section 3.3.5 explains the details of the genetic programming algorithm that evolves the heuristics, such as the parameters and the fitness function used to rate the heuristics' performance.

3.3.1 How The Heuristic is Applied

We obtain a fitness value for a heuristic by evaluating the quality of the solution it creates. This section explains how a heuristic constructs a solution. After reading this section, the reader may also find section 3.3.3 useful, which gives a detailed example of the packing process.

The heuristic operates within a fixed framework, shown in algorithm 1. For each

piece, the framework iteratively applies the heuristic to each open bin in the same way as first-fit iterates through the bins. For each bin, the heuristic returns a value (essentially a score for the bin and piece combination), and if the value is greater than zero then the piece is placed in that bin. If the value is less than or equal to zero then the next bin is considered. After a piece is placed, the cycle repeats with the next piece. Therefore, within this framework, we evolve the mechanism which makes the choice of whether to commit to a bin or move on to consider the next bin.

Algorithm 1 Pseudo code of the framework within which the heuristics are applied in the experiments of chapter 3. The input is a list L of pieces to be packed

```
1: initialise partial solution s with one empty bin
2: for each piece p in L do
3:    for each bin b in s do
4:       heuristic_output = evaluateHeuristic(p, fullness of i, capacity of i)
5:       if heuristic_output > 0 then
6:          place piece p in bin i and break
7:       end if
8:    end for
9:    if piece was placed in the empty bin then
10:      add a new empty bin to s
11:   end if
12: end for
```

It should be noted that we do not constrain the heuristic to only consider bins with enough room for the current piece. In other words, as the heuristic iterates through the bins, if the heuristic finds a bin that the piece cannot fit into, it must still make a decision as to whether to put the piece into the bin. The heuristic is permitted to put the piece into such a bin, and if it does then this produces an 'illegal' solution. Therefore, to produce a *legal* solution, the heuristic must take into account that the current piece is too large and return a value of zero or less. An illegal solution is also produced if, for any one piece, the heuristic never returns a value greater than zero. In such a case, the piece will not be placed in any bin, even the empty bin which is always made available. The concept of illegal solutions is explained further in section 3.3.4. While a heuristic is *permitted* to produce an illegal solution, the heuristic's fitness is heavily penalised if it does (see section

TABLE 3.1: The functions and terminals used in chapter 3, and descriptions of the values they return

	Symbol	Arguments	Description
Functions	$+$	2	Add
	$-$	2	Subtract
	$*$	2	Multiply
	%	2	Protected divide function. A division by zero will return 1
	$\leq$	2	Tests if the first argument is less than or equal to the second argument. Returns 1 if true, -1 if false
	A	1	Returns the absolute value of the argument
Terminals	F	0	Returns the sum of the pieces already in the bin
	C	0	Returns the bin capacity
	S	0	Returns the size of the current piece

3.3.5). This provides a selection pressure towards heuristics which produce legal solutions.

3.3.2 Structure of the Heuristics

The individuals in the population are heuristics that have a tree structure. This is the traditional structure for individuals in a genetic programming population. The leaf nodes of a tree are referred to as 'terminals' and they take a value depending on the state of the problem. The internal nodes of a tree are referred to as 'functions', and they perform operations on the values supplied by the terminals. For a full explanation of the role of functions and terminals, refer back to section 2.7.1.

For a heuristic to produce a solution, it considers each piece in sequence, and decides which bin to pack the current piece into. It chooses a bin by operating within a fixed framework (shown in algorithm 1), which applies the heuristic to each bin in sequence. When the heuristic is applied to a bin, its terminal nodes each acquire a value, dynamically determined by the properties of the current bin and the current piece. The heuristic returns one numerical result, by evaluating its function nodes using the values supplied by its terminal nodes.

The functions and terminals are shown in table 3.1. The function set includes the four basic arithmetic operators, along with a 'less-than-or-equal-to' operator, and a function that returns the absolute value of its input. There are three terminals, 'F', 'S', and 'C'. F

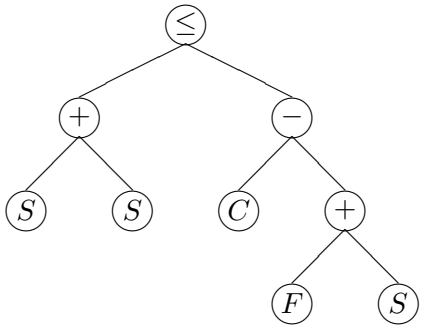

FIGURE 3.5: An example heuristic used in the analysis in section 3.3.3, to pack the pieces in figures 3.6 to 3.9

represents the total size of the pieces already in the bin, C represents the capacity of the bin, and S represents the size of the current piece.

This function and terminal set is intentionally simple, but aims to provide all of the information that a heuristic may need. From this set of components, the heuristic can calculate more complex features, for example the space left in the bin. From the examples given in section 3.3.4, one can see that just using the bin capacity terminal does not provide enough information to intelligently make a decision on where to place the piece. The heuristic must evolve to combine information about both the piece and the bin, in order to ensure the constraints of the bin packing problem are adhered to.

3.3.3 Simple Example of a Heuristic Constructing a Solution

This section provides a step-by-step analysis of a heuristic (shown in figure 3.5) performing part of the packing process of a simple problem instance. It is more informative to begin the example with a partial solution, as it makes more explicit the ability of the terminal nodes to obtain different values as the heuristic is applied to different bins. Therefore, the example begins with four pieces already packed, and four pieces which remain to be packed, as can be seen in figure 3.6. Figures 3.7-3.9 help explain the process of packing the first three of those pieces, using the heuristic shown in figure 3.5. All of the bins have a capacity of 150 in this example. The reader may wish to compare the results of this heuristic with the results of the first-fit heuristic when packing the same instance, shown in section 3.2 and figures 3.1-3.4. The example begins with the partial solution shown in figure 3.6, the heuristic will first choose a bin for the piece of size 70. To do this, the heuristic is applied to each bin in turn, and returns a value for each. This is shown in figure 3.7. The first

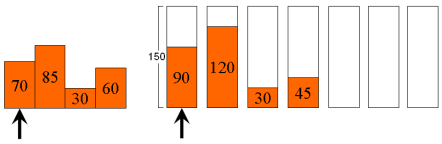

FIGURE 3.6: A partial solution. We will use the heuristic in figure 3.5 to pack the four remaining pieces

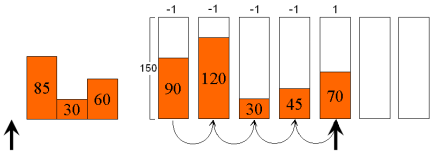

FIGURE 3.7: The piece of size 70 is packed into the first bin for which the heuristic returns a positive number, the fifth bin

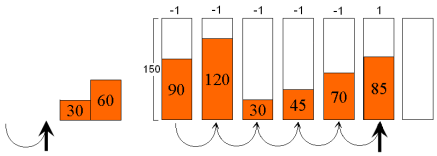

FIGURE 3.8: The piece of size 85 is packed into the sixth bin

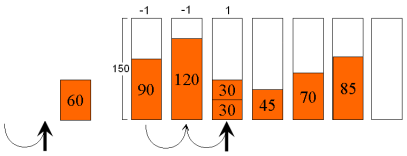

FIGURE 3.9: The piece of size 30 is packed into the third bin

bin has a fullness of 90, and a capacity of 150. The heuristic is shown in equation 3.1 in linear form, and in equation 3.2 with the values for the terminal nodes substituted in. The inequality evaluates to false, so a value of -1 is returned by this node (see the description of $\leq$ in table 3.1). This is the value returned by the heuristic, since the $\leq$ node is at the root of the tree.

$$(S + S) \leq (C - (F + S)) \tag{3.1}$$

$$(70 + 70) \leq (150 - (90 + 70)) \tag{3.2}$$

The value returned by the heuristic is not greater than zero, so the heuristic moves on to the next bin and is evaluated in the same way, the value of the F terminal node will be 120 this time, however, because this bin has a different fullness. The right hand side of the inequality will be even less than for the first bin, because of this increase in the value of F. So the inequality will still not hold, and the heuristic will still return -1.

In fact, the first time that the heuristic returns a value of greater than zero is at bin five, which is empty. The value of zero for the F terminal node means that the inequality evaluates to true, and so it returns 1. The piece is put into this bin, as shown in figure 3.7 and the heuristic moves on to the next piece.

Figure 3.8 shows the process of packing the second piece, which has a size of 85. The bins are iterated through, and a result is returned by the heuristic for each one. This

time, the value of the S terminal will be different, because it is a different piece that is being packed. The sixth bin is the first bin for which the heuristic returns a value greater than zero, and so the piece is put into this bin. It is obvious so far that this heuristic is different to first fit. This is because it often decides not to put a piece into a bin that clearly contains enough space for it, as if it is waiting for a better bin.

When we pack the third piece with the heuristic, it decides to put it into the third bin, which already contains a piece. This process is shown in figure 3.9. At the first bin, when the terminal values have been substituted into equation 3.1 and then simplified, the heuristic is checking the inequality $60 \leq 30$. This evaluates to false, and so -1 is returned. At the second bin, the heuristic also returns -1, because the inequality $60 \leq 0$ is false. At the third bin, the heuristic is checking the inequality $60 \leq 90$, which evaluates to true, and the piece *is* put into this bin. The fourth piece will be packed into bin seven.

From this packing process, it can be shown that this heuristic will never pack a piece into a bin if its size is greater than a third of the free space in the bin. For example, in figures 3.6-3.9, a piece would need to be of size 20 or less for the heuristic to pack it into the first bin. This is because for a bin with 60 units of space left, the piece must be size 20 or lower for the inequality of equation 3.1 to hold.

3.3.4 Heuristics Producing Illegal Solutions

In section 3.3.1, it was stated that a heuristic can produce an illegal solution, and is not artificially constrained not to do so. Through the process of evolution, it must learn the rules of one dimensional bin packing through the feedback it gets from the fitness function. This section provides further clarification on how a heuristic may produce an illegal solution.

All algorithms for bin packing use information about the piece size, the capacity of the bin, and the pieces that have been placed in the bin already, to ensure that they do not violate the hard constraints of the problem. This is the minimum information needed to ensure that the bin does not become overfilled. The heuristics evolved here are not forced to contain all of the terminal nodes that represent these features. They are free to contain any subset of the terminals. This is a decision taken to test if the genetic programming system can select the correct information and use it in the correct way.

If a heuristic does not have the minimum set of terminals that it needs to be able to create a legal solution, then there is a possibility that it may produce an illegal solution.

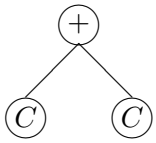

FIGURE 3.10: A heuristic that puts every piece in the first bin

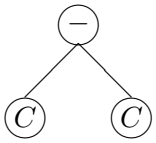

FIGURE 3.11: A heuristic that puts no pieces in any bin

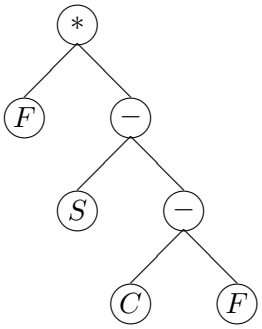

FIGURE 3.12: A heuristic that puts no pieces in any bin

However, a heuristic could also produce an illegal solution if it *does* contain the relevant functions and terminals, but combined in an incorrect way. The following two subsections provide examples of these conditions to further clarify these points.

Two Examples of Heuristics Without All Relevant Terminals

A heuristic only including the capacity terminal (C) cannot possibly make a relevant decision as to whether a piece should be placed into a bin. For example, if the heuristic was $C + C$ (figure 3.10), then the result returned would always be twice the bin capacity. This value will always be greater than zero, and so every piece will be placed into the first bin, which will eventually overfill it. Such a heuristic is permitted to remain in the population, it is not artificially deleted, nor is its choice of bin constrained. However, it will receive a very low fitness value and therefore it will be highly unlikely to be chosen to be a parent of individuals in the next generation.

Similarly, if the heuristic was $C - C$ (figure 3.11), then the result would always be zero. In this case, the piece will not be put into any bin, and the algorithm will iterate through all the available bins and never decide to put the piece into one of them. This also produces an illegal solution because a hard constraint of the problem is that all of the pieces must be packed. Again, the algorithm is not constrained to definitely pack all of the pieces. The heuristic must evolve its own mechanism to ensure that all of the pieces are packed and that no bin is overfilled.

TABLE 3.2: Genetic programming initialisation parameters for chapter 3

Population size	1000
Generations	50
Crossover Proportion	90%
Reproduction Proportion	10%
Maximum depth of initial trees	4
Method of initialising the trees	Grow
Selection Method	Roulette wheel

An Heuristic Combining the Relevant Terminals in an Incorrect Way

Figure 3.12 shows a heuristic containing all the terminals available. The way they are combined, however, means that the heuristic will not put any piece in any bin, because the fullness of the first bin is zero before any pieces are packed, and the final calculation of the heuristic makes it multiply the right hand side of the tree by the fullness. This calculation will always be a multiplication with zero, and therefore the heuristic will never return a value greater than zero.

3.3.5 How the Heuristics are Evolved

Previous sections of this chapter have explained how a heuristic is applied to a problem instance to obtain a solution, and what a heuristic consists of. This section explains how the heuristics themselves are evolved by the genetic programming algorithm. There are three parts to this section. The first part will provide an overview of the genetic programming parameters and the experimental setup. The second part will describe the benchmark data set on which we evolve the heuristics. The third part will describe two fitness functions used to assess the performance of each heuristic.

Experimental Setup

The parameters of the genetic programming run are shown in table 3.2. These values are used because they are standard parameters, which result in a good solution in reasonable time. As stated in chapter 1 on table 1.1, the genetic programming system used for the experiments of this chapter is the author's own code. This is also true for chapter 4, while the subsequent chapters use the ECJ (Evolutionary Computation in Java) package (http://www.cs.gmu.edu/∼eclab/projects/ecj/).

A population size of 1000 is used, with the maximum number of generations set to 50. 90% of a new generation is generated by the crossover operator, and 10% by the reproduction operator. The individuals for these operators are selected via roulette wheel selection.

The relatively small value of four is chosen for the maximum initial depth of the trees, because the trees' size is not limited during the run. The 'grow' method [167] of initialising the trees is used. Fitness proportional (roulette wheel) selection is employed to select individuals from the population for the crossover and reproduction operators. This method selects individuals in proportion to their fitness relative to the rest of the population.

These parameters are intentionally basic. They are standard parameters, most of which are used in [164], which are sufficient to produce the results presented in this chapter. To put this decision in context, the aim of this initial chapter is to discover if a basic genetic programming system can produce human competitive heuristics, and we therefore opt to use a simple configuration initially, then add richness to the parameters once a proof of concept has been established.

For example, the mutation operator is considered in the literature to be beneficial in the majority of cases, and is used in chapters 6 and 7 of the thesis. The 'grow' method of initialising the population is replaced in subsequent chapters by the 'ramped half-and-half' method. Also, parsimony pressures are applied to the population in later chapters, but they are not used in this chapter.

The next section explains that there are 20 instances used in the experiments of this chapter. The results section presents heuristics evolved on one problem instance and heuristics evolved on all 20 instances. During the genetic programming run, each individual is assigned a fitness which is either based on its performance on one instance ('evolved on' one instance), or its performance on all of the instances ('evolved on' all 20 instances).

In the results section, 20 heuristics are evolved in each set of experiments. One run of the genetic programming system always outputs just one heuristic, so therefore 20 runs of the genetic programming system are performed in each set of experiments. Algorithm 2 shows the genetic programming algorithm. Because of available computing resources, 20 heuristics were evolved for each experiment in this chapter. Evolving this many heuristics provides a reliable average performance value. In subsequent chapters, 50 heuristics are evolved for each experiment, as they are evolved on groups of instances only, and not single instances. Therefore we could afford to evolve more heuristics for each group.

Algorithm 2 Pseudo code of the genetic programming algorithm used in the experiments of chapter 3. The input is a set of problem instances I, which could contain just one instance, or it could contain 20 instances. Algorithm 1 is referred to, which details how a heuristic produces a solution. The genetic operators are selected with probabilities detailed in table 3.2

```
1: initialise population
2: for generation g ≤ 50 do
3:    for each individual ind in population do
4:       for each instance i ∈ I do
5:          fitness(ind) += getFitness(algorithm1(ind, i))
6:       end for
7:    end for
8:    initialise new population nextgen
9:    repeat
10:      select genetic operator
11:      select parent(s) with roulette wheel selection
12:      insert child into nextgen
13:   until population nextgen is filled population = nextgen
14: end for
```

The Data Set

The dataset that is used to evolve the heuristics is a set of 20 instances, generated by Falkenauer to test a hybrid grouping genetic algorithm for bin packing [96]. The dataset is available at the OR-Library at *http://people.brunel.ac.uk/∼mastjjb/jeb/orlib/binpackinfo.html*, where the set is referred to as 'binpack0'. Each instance has 120 pieces, with sizes uniformly distributed between 20 and 100 inclusive.

The Fitness Functions

After the heuristic has constructed a solution, its fitness is obtained by evaluating the the quality of the solution that it produced. In this chapter, two fitness functions are investigated, and their performance will be compared in the results section.

The results section presents heuristics evolved on one problem instance, *and* heuristics evolved on all 20 problem instances in the dataset (each heuristic packs 20 instances and its performance is then assessed). The two fitness functions (shown in equations 3.3 and 3.4) obtain a fitness for the former of the two. For the latter, the fitness is obtained by applying the fitness function to each packed instance and summing the results. The total is then assigned as the fitness of the individual.

The 'basic' fitness function is shown in equation 3.3, where B represents the number of bins used, n represents the number of pieces in the instance, S_k represents the size of piece k, and C represents the bin capacity.

$$Fitness = B - \frac{\sum_{k=1}^{n} S_k}{C} \tag{3.3}$$

This fitness function finds the difference between the number of bins used and a trivial theoretical upper bound on the number of bins needed. The upper bound is calculated by summing the piece sizes and dividing by the capacity of the bins, and then rounding the result up to the nearest integer. A fitness closer to zero is better, and a fitness equal to zero means the solution obtained by the heuristic is provably optimal. Illegal solutions (see section 3.3.4) are given a fitness value of 1000, an arbitrarily high value compared to the range of fitness values that a legal solution can have. The worst legal solution is obtained by allocating only one piece to every bin. As there are 120 pieces, a legal solution can never have a fitness greater than 120, which is much lower than 1000.

The second fitness function is shown in equation 3.4, where: n = number of bins.

m = number of pieces. w_j = size (weight) of piece j. $x_{ij} = 1$ if piece j is in bin i and 0 otherwise. C = bin capacity.

$$Fitness = 1 - \left(\frac{\sum_{i=1}^{n} \left(\frac{\sum_{j=1}^{m} w_j x_{ij}}{C} \right)^2}{n} \right) \tag{3.4}$$

This fitness function is taken from [97]. It puts a premium on bins that are filled completely or nearly so. Importantly, the fitness function is designed to avoid the problem of plateaus in the search space, that occur when the fitness function does not discriminate between heuristics that use the same number of bins. Solutions with lower fitness are better, and a fitness of 1000 is assigned to any heuristic that produces an illegal solution.

3.4 Results

This section presents the results of the experiments performed with the genetic programming hyper-heuristic described in this chapter. Section 3.4.1 presents some evolved heuristics obtained during preliminary experiments, which approximate the performance of first-fit. Section 3.4.2 explains the two statistical tests that are used to validate the two categories of results. Section 3.4.3 analyses all of the heuristics evolved with the basic fitness function. This can be compared to section 3.4.4, which analyses the results of the heuristics evolved with the *second* fitness function. Section 3.4.5 presents the results of heuristics evolved without the $\leq$ function in the function set.

3.4.1 Individual Results

This section presents individual heuristics obtained in the first set of experiments, using the basic fitness function. They are intended to provide the reader with some examples of what the evolved heuristics look like, and how similar they are to the human created first-fit heuristic.

Small Trees

Figures 3.13-3.16 show four individuals evolved from four different runs (labelled A to D), which illustrate that simple programs can be found by the genetic programming system, and that the system is versatile enough to produce many different ways of expressing the same heuristic, including some ways that would perhaps not be thought of immediately by

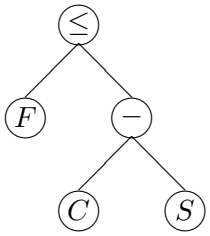

FIGURE 3.13: Tree A

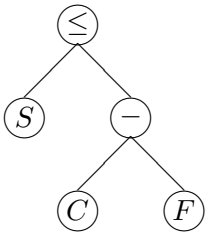

FIGURE 3.14: Tree B

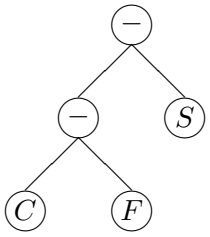

FIGURE 3.15: Tree C

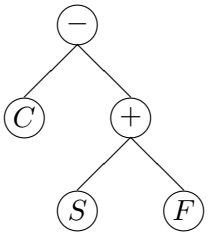

FIGURE 3.16: Tree D

a human programmer given the same task. They all result in a legal solution because a piece is never put in a bin when it is larger than the space left in the bin.

All four individuals perform much the same function. However, trees C and D are slightly different to A and B. Trees A and B both return a negative number if the piece size is greater than the space left in the bin, and a positive number if not. They, therefore, implement the first-fit heuristic. Trees C and D perform the same way, apart from the case when the piece size is the *same* as the space left in the bin. Zero is returned in this case, because the '$\leq$' terminal returns zero when its two arguments are equal. This means that the greater-than-zero condition (shown in algorithm 1) is not met, and the piece is not placed in the bin. So C and D are individuals that do not implement the first-fit heuristic, but their functionality is highly similar.

It is interesting to analyse why heuristics C and D were *best-of-run* individuals, when theoretically superior heuristics, such as A and B, seem easy to evolve because of their relatively small size. This can be explained by imagining that if heuristic C is found first in its run, then it is stored as the *best-of-run* individual so far. In subsequent generations, heuristics like A and B most probably *were* produced, but with the data sets used here they do not produce a solution which used less bins. They, therefore, would not replace heuristic

C as the *best-of-run* individual, because they would not have a better fitness according to the basic fitness function.

A possible reason that a better result is not gained by using heuristics A and B, is because the bin capacity is large and the piece sizes are such that it is uncommon for a bin to be filled exactly. Therefore, the fact that the piece will not be put in the bin if it exactly fills the bin barely affects the assignment of pieces to bins. Therefore, the solutions produced by both heuristics are almost the same, and they receive the same fitness.

Plateaus in the search space (where many different heuristics map to the same fitness) also mean that code-bloat [22] was not especially problematic to the genetic programming algorithm in the runs which produced figures 3.13-3.16. They were found in the early generations when the average tree size was small, they were stored as the best so far, and then were not surpassed by individuals found in later generations. The average complexity of the trees only increases in later generations. Therefore, if a good heuristic is found early in the run, it is more likely to be a simple and easy to understand tree than if it is found in a later generation. The next section presents and describes two heuristics which were stored in later generations of a run, and are more complex then those presented in this section.

More Complex Trees

We present two more examples of heuristics created during different runs, which are more complex than the four individuals presented in figures 3.13-3.16. These individuals are of interest because they are quite far removed from any program that a human programmer would create for the same task. Both are *best-of-run* individuals.

Tree Structure 1

Figure 3.17 has the same functionality as the first-fit heuristic, because F is always less than or equal to C. For this reason, 1 will always be returned by the left branch. In general, if inequality 3.5 is satisfied, then the piece fits into the bin. The right hand branch will return 0 or less if this is the case. Therefore, the result returned by the whole tree will be 1 or greater, and the piece will be put in the bin.

$$(F + (S - C)) <= 0 \tag{3.5}$$

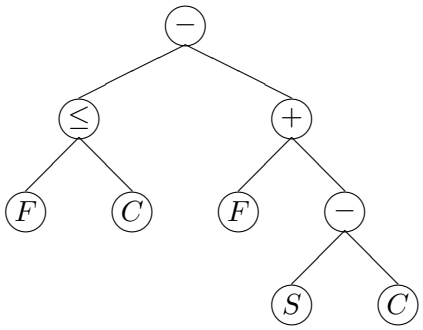

FIGURE 3.17: Tree E

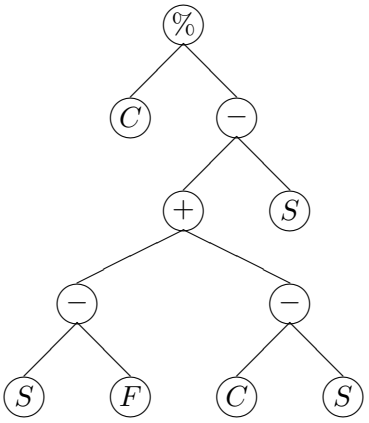

FIGURE 3.18: Tree F

Tree Structure 2

Figure 3.18 shows a tree which also implements the first-fit heuristic. The heuristic's right hand branch (from the first subtract node downwards) can be simplified to expression 3.6.

$$(C - F) - S \tag{3.6}$$

This branch will return 0 or greater if the piece fits, or else it will return a negative number, as $C - F$ will be smaller than S. Since C is always a positive integer number, C divided by a positive number returns a positive number, and so the piece is placed in the bin in such a case. On the other hand, C divided by a negative number returns a negative number, and so if the right hand branch evaluates to a negative number then the piece is not placed in the bin.

The most interesting feature of figure 3.18, however, is the way that the 'protected divide' node at the root fixes the problem that figures 3.15 and 3.16 have. Recall that in those trees, when the piece fits exactly, zero is returned, and therefore the 'greater-than-zero' condition, shown in the pseudocode (algorithm 1), is not satisfied, and the piece is not placed in the bin. The protected divide function at the root of figure 3.18 means that when the piece fits exactly, 1 is returned as it is a division by zero. So the problems in figures 3.15 and 3.16 have been solved in figure 3.18.

This functionality was not the reason the protected divide function was defined in this way. The reason it was defined this way was for closure of the function set, as explained in [164]. This is, therefore, an example of how the evolutionary process can use

combinations of functions and terminals in ways not originally envisaged by the human programmer that supplied them.

3.4.2 Statistical Tests

This section explains the two statistical tests used to analyse the results in sections 3.4.3-3.4.5.

In each of the sections 3.4.3-3.4.5, there are two categories of heuristics for which results are reported. The first category consists of heuristics which are each evolved on all 20 instances. Each section presents the results of 20 heuristics evolved from this category. The paired difference t-test is used to analyse this category of results, to see if each evolved heuristic performs better than first-fit or best-fit. The second category consists of heuristics evolved for just one given instance. Each section presents the results of 400 heuristics, as 20 heuristics are evolved on each of the 20 instances. The paired-difference t-test is used to analyse this category, to see if the genetic programming evolves heuristics that are better than first-fit and best-fit on average.

The t-test

For each individual instance, 20 heuristics are evolved. The results of these 20 heuristics are compared to the result of first-fit and best-fit on the same instance with a t-test. For example, a set of 20 evolved heuristics produces a set of 20 results on one instance. First-fit obtains one result on one instance, the number of bins it needs. The null hypothesis (H_0) of the t-test is that the set of 20 results has a mean equal to the result of first-fit. The aim of the t-test is to disprove the null hypothesis, meaning we could conclude that the 20 results are taken from a population with a mean different to the result of first-fit. We perform a one-tailed t-test with a confidence level of 95% to determine if the heuristics we evolve come from an underlying distribution with a mean less (i.e. better) than the result of first-fit.

If the mean is worse, then we also want to test if the heuristics we evolve are *significantly* worse than first-fit. Therefore, when the mean is greater (i.e. worse), then we perform a one-tailed t-test to see if the result is *significantly* greater. The results of the heuristics are compared to the result of best-fit in the same way. The results are presented by stating whether a set of 20 heuristics is significantly better or significantly worse than the human-created heuristic, or if there is no significant difference either way.

The paired-difference t-test

The 20 heuristics evolved on *all* 20 instances are each compared to first-fit and best-fit with a paired-difference t-test. When comparing to first-fit, the 20 results the evolved heuristic obtained on the 20 instances are compared to the 20 results that first-fit obtained. The results are paired because they depend on the specific instance they were obtained on, and it is, therefore, meaningless to compare two results from different instances. The null hypothesis (H_0) is that the mean difference between the two sets of results is zero. The aim of the test is to disprove the null hypothesis, and show that the mean difference between the 20 pairs of results is greater than zero. If the mean difference is greater than zero then the evolved heuristic will be shown to be worse than first fit, because the heuristic that uses more bins is worse. The test is one-tailed, because we are testing if an evolved heuristic is *worse* than first-fit, we are not testing if an evolved heuristic is just *different* to first-fit. Results are taken to be significant at the 95% confidence level. The 20 heuristics are then compared to best-fit in the same way.

3.4.3 Comparison with First-Fit and Best-Fit, Employing the Basic Fitness Function

Evolving heuristics for all 20 instances

Table 3.3 summarises the results of 20 heuristics each evolved on all 20 problem instances, using the basic fitness function (see equation 3.3). They must, therefore, perform well on all of the instances if they are to survive in the population. The top two rows of the table show the mean and standard deviation of the number of bins needed by the heuristics in total, to pack all of the 20 instances.

The third and fourth rows show the performance of first-fit and best-fit on the same instances. This demonstrates that the evolved heuristics have a similar performance to first-fit. However, they do not perform as well as best-fit, which uses six bins less than first-fit. Fifteen heuristics used 1044 bins in total over the 20 instances, 4 heuristics used 1043, and one heuristic used 1042 bins. Note though that, the difference in performance is not statistically different, and, therefore, we cannot say that the evolved heuristics are superior. As stated in section 3.4.2, we compare each of the evolved heuristics to first-fit and best-fit, on all 20 instances, with a paired-difference t-test. The final two rows of table 3.3 show a summary of these statistical comparisons in terms of the number worse than first-fit or best-fit. The 20 evolved heuristics are not statistically worse than first-fit, and

TABLE 3.3: The performance of 20 heuristics, each evolved on all 20 instances using the basic fitness function which is shown in equation 3.3, section 3.3.5

Performance of 20 evolved heuristics	Average total number of bins used	1043.7
	Standard deviation	0.571
Performance of other heuristics	Performance of first-fit	1044
	Performance of best-fit	1038
Statistical analysis	Number not statistically worse than first-fit	20
	Number not statistically worse than best-fit	1

one is not statistically worse than best-fit. These results show that the evolved heuristics consistently have the same performance as the first-fit heuristic.

Evolving heuristics for one of the 20 instances

There are 20 heuristics evolved on each individual instance, making 400 in total, as there are 20 instances. Table 3.4 is split into two sections to display the results. The upper section displays the results from the heuristics evolved on instances 1-10, and the lower section displays instances 11-20. In each section, the first row shows the average number of bins used by the 20 heuristics evolved on each instance, and the standard deviation of the distribution. The third and fourth rows of the table show the results of first-fit and best-fit on each instance as a comparison.

The fifth and sixth rows show the results of statistical comparisons (see section 3.4.2) between the distribution of results of the 20 heuristics, and the result of first-fit or best-fit. If we look at the fifth row, a '$\surd$' means that for the instance, the genetic programming hyper-heuristic generates heuristics better than first-fit. A '$\times$' means that the evolved heuristics are worse than first-fit. A '=' symbol means that there is no significant difference between the heuristics generated by the genetic programming system and first-fit.

Therefore, table 3.4 shows that, for eight of the individual instances, the genetic programming hyper-heuristic creates heuristics which are statistically better than first-fit. On the remaining twelve instances, there is no significant difference. When compared with best-fit, the hyper-heuristic creates better heuristics for five instances, and worse heuristics for six instances.

To comment on the difference between evolving on one instance or all 20 instances, it seems harder for the genetic programming system to evolve better heuristics over 20 instances than when they are evolved over one instance. This can be seen by the result

TABLE 3.4: The performance of 20 heuristics each evolved on one instance, using the basic fitness function which is shown in equation 3.3, section 3.3.5

Instance	1	2	3	4	5	6	7	8	9	10
Average number of bins used	50.0	51.0	48.0	51.95	52.0	51.7	50.9	52.0	53.4	49.0
Standard deviation	0	0	0	0.22	0	0.47	0.31	0	0.50	0
Performance of first-fit	50	51	48	52	52	52	51	52	54	49
Performance of best-fit	50	51	48	53	52	52	52	52	53	48
First-fit statistical comparison	=	=	=	=	=	√	=	=	√	=
Best-fit statistical comparison	=	=	=	√	=	√	√	=	×	×
Instance	11	12	13	14	15	16	17	18	19	20
Average number of bins used	55.1	51.6	51.9	50.95	52.6	52.0	55.4	56.0	51.95	51.65
Standard deviation	0.31	0.50	0.31	0.22	0.50	0	0.50	0	0.22	0.49
Performance of first-fit	56	52	52	51	53	53	56	56	52	52
Performance of best-fit	55	51	51	51	53	52	55	56	51	52
First-fit statistical comparison	√	√	=	=	√	√	√	=	=	√
Best-fit statistical comparison	=	×	×	=	√	=	×	=	×	√

that no heuristic evolved over 20 instances is significantly better than first-fit, but there *are* some heuristics evolved on individual instances which are significantly better than best-fit.

3.4.4 Comparison with First-Fit and Best-Fit, Employing the Second Fitness Function

This section takes the same format as section 3.4.3, and presents the results obtained when the second fitness function is used to guide the evolution, instead of the basic fitness function.

Evolving heuristics for all 20 instances

Table 3.5 shows the results of the 20 heuristics evolved on all 20 instances. This set of 20 heuristics has a slightly better mean performance than the set evolved using the basic fitness function, as can be seen by comparing rows two and four of table 3.5. However, this

TABLE 3.5: The performance of 20 heuristics each evolved on all 20 instances using the second fitness function which is shown in equation 3.4, section 3.3.5

Performance of 20 evolved heuristics	Average number of bins used	1043.45
	Standard deviation	0.605
Performance of other heuristics	Using the basic fitness function	1043.7
	Performance of first-fit	1044
	Performance of best-fit	1038
Statistical analysis	Number not statistically worse than first-fit	20
	Number not statistically worse than best-fit	1

difference is not found to be statistically significant when a two-sample t-test is performed on the two sets of 20 performance values. Using the paired-difference t-test, there is no significant difference found between the performance of any evolved heuristic and first-fit. Also, best-fit is statistically better than 19 of the evolved heuristics. These results suggest that changing to the second fitness function does not make a significant difference when evolving heuristics on all 20 instances.

Evolving heuristics for one of the 20 instances

Table 3.6 shows the results for the heuristics evolved on one instance with the second fitness function. The average number of bins used by the heuristics evolved with the basic fitness function (see section 3.4.3) is given in row four as a comparison. In contrast to table 3.4, the results of first-fit and best-fit are omitted from table 3.6 because they are given in table 3.4. Ultimately, it is the statistical comparison that matters most rather than the actual values that are compared.

The statistical comparisons are given in rows five and six of table 3.6. They show that with the second fitness function, the hyper-heuristic evolves heuristics which are significantly better than first-fit on nine instances. This is one more than for the basic fitness function, where this was true for eight instances. This could suggest that the second fitness function makes a positive difference, but we do not claim that this is a conclusive result.

Comparing with best fit, there are five instances where the evolved heuristics are better, and seven instances where they are worse. This can be compared to the results of the basic fitness function, where the evolved heuristics are worse on six of the instances. This could suggest that the second fitness function makes a negative difference to the evolution process. However, overall, these results do not seem to be different enough to be able to

TABLE 3.6: The performance of 20 heuristics each evolved on one instance, using the second fitness function which is shown in equation 3.4, section 3.3.5

Instance	1	2	3	4	5	6	7	8	9	10
Average number of bins used	50.0	51.0	48.0	51.95	52.0	51.8	51.0	52.0	53.4	49.0
Standard deviation	0	0	0	0.22	0	0.41	0	0	0.50	0
Evolved with basic fitness function	50.0	51.0	48.0	51.95	52.0	51.7	50.9	52.0	53.4	49.0
First-fit statistical comparison	=	=	=	=	=	√	=	=	√	=
Best-fit statistical comparison	=	=	=	√	=	√	√	=	×	×
Instance	11	12	13	14	15	16	17	18	19	20
Average number of bins used	55.4	51.55	52.0	51.0	52.75	52.0	55.4	55.9	51.85	51.65
Standard deviation	0.50	0.51	0	0	0.44	0	0.50	0.31	0.37	0.49
Evolved with basic fitness function	55.1	51.6	51.9	50.95	52.6	52.0	55.4	56.0	51.95	51.65
First-fit statistical comparison	√	√	=	=	√	√	√	=	√	√
Best-fit statistical comparison	×	×	×	=	√	=	×	=	×	√

conclude that the second fitness function makes any difference to the evolution of heuristics on these individual problems.

3.4.5 Comparison with First-Fit and Best-Fit, Removing the $\leq$ Function

This section takes the same format as the previous two. The second fitness function is used for this set of results, but this time the $\leq$ function is removed from the function set. This function is different to the others in some respects, because it returns one of only three values, -1, 0, or 1. This function is potentially only useful at the root of the tree, where its result will clearly fall on either side of zero. However, potentially, a lot of information could be lost as calculations performed further down the tree are condensed to either -1, 0, or 1. Therefore, it is possible that better results can be obtained without this function.

Evolving heuristics for all 20 instances

Table 3.7 shows the results of the 20 heuristics evolved on all 20 instances without the $\leq$ function. The table compares these results to those previously reported in sections

TABLE 3.7: The performance of 20 heuristics each evolved on all 20 instances, without the $\leq$ function included in the function set, and with the second fitness function, which is shown in equation 3.4, section 3.3.5

Performance of 20 evolved heuristics	Average number of bins used	1043.05
	Standard deviation	1.146
Performance of other heuristics	With the $\leq$ function and the second fitness function	1043.45
	With the $\leq$ function and the basic fitness function	1043.7
	Performance of first-fit	1044
	Performance of best-fit	1038
Statistical analysis	Number not statistically worse than first-fit	20
	Number not statistically worse than best-fit	4

3.4.3 and 3.4.4. It appears that removing the $\leq$ function from the function set has had little impact, although the average number of bins used by the 20 heuristics has decreased slightly. However, this difference is not significant when a two-sample t-test is applied to both sets of results. None of these evolved heuristics are statistically worse than first-fit (similar to the heuristics evolved *with* the $\leq$ function). This time, however, one heuristic is evolved that is statistically *better* than first-fit (verified by a one-tailed t-test to see if the heuristic is better rather than worse). Also, in contrast to the results of tables 3.3 and 3.5, there are four heuristics which are found not to have worse performance than best-fit. In conclusion, it appears that removing the $\leq$ function has a positive impact on the evolution of heuristics each trained on all 20 problem instances.

Evolving heuristics for one of the 20 instances

Table 3.8 shows the results of the 20 heuristics evolved for each instance without the $\leq$ function included in the function set. The second fitness function is used in this experiment. The results from the previous sections 3.4.3 and 3.4.4 (which included $\leq$) are shown in the fourth and fifth rows as a comparison. The statistical comparisons with first-fit and best-fit are given in rows six and seven.

The statistical tests show that for twelve instances, the genetic programming produces heuristics significantly better than first-fit. This is better than the result with $\leq$ included, where this was true for nine instances. Also, the genetic programming produces heuristics which are better than best-fit for seven of the instances, and worse than best-fit for five of the instances. This compares well with the results of the heuristics evolved with $\leq$ included (see section 3.4.4), where five were better and seven were worse. Therefore, it

appears that removing the $\leq$ function has a positive impact on the evolution of heuristics for these problem instances.

3.5 Conclusion

This chapter has shown that a genetic programming hyper-heuristic system can consistently evolve heuristics of at least the same quality as the human-designed first-fit heuristic. This is the case whether the heuristics are evolved on 20 instances or just one instance. It has also been shown that, when evolving heuristics for certain single instances, the hyper-heuristic can generate heuristics better than best-fit. However, no heuristic is evolved over all 20 instances that is better than best-fit. Some are evolved which statistically perform as well as best-fit, but they always use 4-5 bins more in total, with this result just not being significant enough over the 20 instances to be able to disprove the null hypothesis. Best-fit seems to be a more general heuristic than any that can be evolved with the hyper-heuristic system presented in this chapter.

Two fitness functions have been investigated in this chapter, a basic one, and a more complex version which takes more information into account than just the number of bins used. It is possible that the basic fitness function could cause large plateaus in the search space, preventing better heuristics from being stored over technically inferior ones (see section 3.4.1), and making the the search for an optimal heuristic almost random [97]. This could be a difficulty as the problem instances we evolve heuristics for become more complex. There was no significant difference in the performance of the two fitness functions, but the second one will be used in future chapters because it resulted in a slightly improved mean performance value, and it addresses the problem of plateaus in the search space to some extent.

An experiment was also performed to test if the $\leq$ function is necessary in the genetic programming function set. The results of section 3.4.5 show that it is not necessary to include this function, and in fact the results improve when it is taken out. Recall that a piece is placed in a bin if a greater than zero condition is satisfied (see algorithm 1). The $\leq$ function intuitively seems useful because its result of -1, 0 or 1 falls neatly on either side of this inequality. However, it may oversimplify large sections of a tree, keeping the heuristic's functionality quite simple. When it is removed from the function set, this potential simplification is removed, and the five remaining functions are evidently

TABLE 3.8: The performance of 20 heuristics each evolved on one instance, without the $\leq$ function included in the function set, and with the second fitness function, which is shown in equation 3.4, section 3.3.5

Instance	1	2	3	4	5	6	7	8	9	10
Average number of bins used	50.0	51.0	48.0	51.95	51.95	51.6	50.75	51.85	53.05	48.95
Standard deviation	0	0	0	0.22	0.22	0.50	0.44	0.37	0.22	0.22
With $\leq$ function and second fitness function	50.0	51.0	48.0	51.95	52.0	51.8	51.0	52.0	53.4	49.0
With $\leq$ function and basic fitness function	50.0	51.0	48.0	51.95	52.0	51.7	50.9	52.0	53.4	49.0
First-fit statistical comparison	=	=	=	=	=	$\surd$	$\surd$	$\surd$	$\surd$	=
Best-fit statistical comparison	=	=	=	$\surd$	=	$\surd$	$\surd$	$\surd$	=	$\times$
Instance	11	12	13	14	15	16	17	18	19	20
Average number of bins used	55.15	51.45	51.9	50.9	52.55	51.8	55.2	55.85	51.65	51.40
Standard deviation	0.49	0.51	0.31	0.31	0.51	0.62	0.41	0.37	0.49	0.50
With $\leq$ function and second fitness function	55.4	51.55	52.0	51.0	52.75	52.0	55.4	55.9	51.85	51.65
With $\leq$ function and basic fitness function	55.1	51.6	51.9	50.95	52.6	52.0	55.4	56.0	51.95	51.65
First-fit statistical comparison	$\surd$	$\surd$	=	=	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Best-fit statistical comparison	=	$\times$	$\times$	=	$\surd$	=	$\times$	$\surd$	$\times$	$\surd$

still sufficient to be able to construct a good quality heuristic.

There are some instances from the set of 20 for which we can conclude that either best-fit performs very well, or that the genetic programming finds it more difficult to produce heuristics of the same quality. For example, for instances 10, 12, 13, 17 and 19, the genetic programming statistically produces heuristics worse than best-fit in all three main experiments in this chapter (see tables 3.4, 3.6 and 3.8).

Recall that the human designed heuristics for the online bin packing problem (explained in table 2.1 in section 2.3.2) operate by either considering all the bins before making a choice (for example, best-fit), or they iterate through each bin in turn and make a choice immediately after considering the current bin (for example, first-fit). In this chapter, we have used the latter as a framework and evolved the heuristic which makes the decision to pack the piece or move on to the next bin. Better results may be obtained if we use the former framework, and evolve a heuristic which decides where to place the piece after iterating through and evaluating *all* of the bins.

The next chapter investigates the question of determining if using such a framework, based on the operation of best-fit, can enable the heuristics to achieve better results. Importantly, we also test to see if these evolved heuristics can be successfully used on new problem instances after they have been evolved.

Chapter 4

The Reusability of Evolved Online Bin Packing Heuristics

4.1 Introduction

In chapter 3, the genetic programming hyper-heuristic evolved heuristics on single instances and it was shown that they performed better than the best-fit heuristic. However, the heuristics evolved over 20 instances did not perform better than best-fit. At the end of chapter 3, we concluded that this was because of the framework used to apply the heuristics. Therefore, a different framework is employed in this chapter.

In the previous chapter, the heuristic chooses a bin for a piece by returning a value which is greater than zero, and then the subsequent bins are not considered. In the present chapter, the framework applies the heuristic to *all* the bins, returning a value for each, and the bin that obtains the highest value is the one chosen by the heuristic for the current piece. One aim of this chapter is to show that, by employing this framework, the quality of the evolved heuristics are consistently as good as best-fit. Another aim of the chapter is to evolve heuristics which are applicable to new (unseen) problem instances after the heuristic has been evolved on a given set of instances. Therefore, we use 20 instances to train every heuristic evolved in this chapter. We investigate three areas of the behaviour of the evolved heuristics on unseen instances. The three areas are the heuristics' overall quality, how specialised they are, and their robustness.

The quality of the heuristics is ascertained by comparing the performance of each evolved heuristic to best-fit, on 20 new problem instances similar to those that they were evolved on. The extent of any trade-off between specialisation and generalisation is in-

vestigated by creating seven classes of problem instances, each defined by their piece size distribution. There is one general class, two intermediate classes, and four narrow classes, forming a hierarchy. The results will show whether a heuristic can be evolved to be specialised to a narrow piece size distribution, and how this affects its performance on other classes.

There are circumstances where heuristics perform well on new instances from the class they were evolved on, but produce illegal results on instances from other classes. Illegal results are possible because the heuristics are not artificially constrained to adhere to the hard constraints of the problem. In this chapter, we test the robustness of the evolved heuristics by applying them to instances from classes other than those they were evolved on, and show the circumstances where illegal results occur. This issue is important because it affects the trust that can be placed in heuristics created by a computer system as opposed to those created by a human.

In summary, the main aims of this chapter are twofold. They are, firstly, to investigate if we can obtain better heuristics with a framework different to that of chapter 3 and secondly, to investigate the circumstances under which the heuristics can be reused on new problems.

Section 4.2 explains the best-fit heuristic, which is the main performance benchmark in this chapter. Section 4.3 explains the framework that the heuristic operates within, the functions and terminals used, an example of the packing process, and the genetic programming parameters that are used to evolve the heuristics. Section 4.4 presents the results, and section 4.5 presents the conclusions.

4.2 The Best-Fit Heuristic

The best-fit heuristic is explained here because we use it as the main benchmark heuristic to judge the performance of the heuristics evolved in this chapter. It also serves as inspiration for the fixed framework that the evolved heuristics operate within, similar to the way first-fit inspired the framework for the heuristics in the last chapter. Therefore, it is important to understand how best-fit works before reading the rest of this chapter.

We explain the operation of best-fit through a simple example, starting from a partial solution identical to the one that was used to explain first-fit in section 3.2. A comparison of the two solutions shows that best-fit is superior on this small example. Note

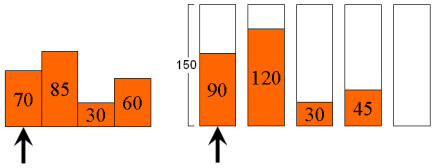

FIGURE 4.1: The start of the example of section 4.2, the piece of size 70 will be packed next by the best-fit heuristic

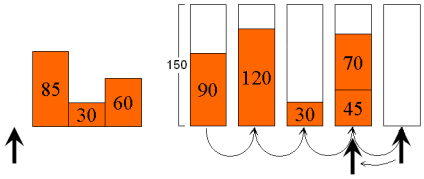

FIGURE 4.2: The piece of size 70 is packed into the fourth bin because it has the least free space of all the bins that the piece fits into

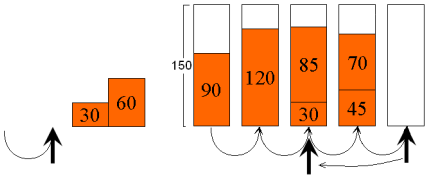

FIGURE 4.3: The next piece is packed into the third bin after all the bins are considered

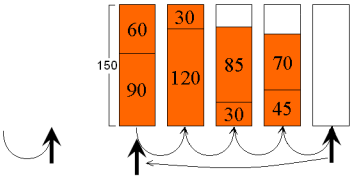

FIGURE 4.4: The final two pieces are packed into the second and first bins

though that the results of chapter 3 show that best-fit is better than first-fit on some instances, but that the reverse is true on other instances. First-fit also has the advantage of being quicker, as it considers less bins in the average case. However, best-fit does seem to perform better in practice, as it did overall on the instances in chapter 3. Theoretical results have shown that no other online heuristic is superior in terms of both the average case and worst case [162].

Figure 4.1 shows a partial solution, with four pieces to pack, and four pieces already packed. The best-fit heuristic begins with the first piece in the list (as it is an online heuristic). To find a bin for this piece, the heuristic iterates through *all* the bins. This is in contrast to the first-fit heuristic, which does not have to consider all of the bins, only those up to the bin that it finally decides upon. Of the bins that the piece fits into, best-fit places the piece into the bin with the least free space. This process is shown in figure 4.2, where the piece of size 70 is placed into the fourth bin. This is because the piece cannot fit into the first two bins, which have a fullness of 90 and 120. The piece fits into the third, fourth and fifth bins, but the fourth bin (with a fullness of 45) has the least free space.

The next piece is then considered, which has a size of 85. The process of packing

this piece is shown in figure 4.3. It does not fit onto the first two bins, or the fourth bin, but it does fit into the third and fifth bins. The third bin has less free space, so the piece is placed there. The best-fit heuristic does not put a piece into an empty bin if it can fit into any partially filled bin. Figure 4.4 shows that the final two pieces are packed into the second and first bins, because they fit exactly.

This heuristic is the inspiration for the framework used to apply the heuristics in this chapter. Within this new framework, all of the bins are considered by the heuristic, just as all of the bins have to be considered by best-fit before it can determine which bin has the least free space. Section 4.3.1 explains this framework further.

4.3 Methodology

4.3.1 How The Heuristic is Applied

The genetic programming hyper-heuristic maintains a population of heuristics represented as tree structures. To create each new generation of heuristics from the previous generation, we must obtain a fitness value for each heuristic. The fitness value is obtained by evaluating the quality of the solution that the heuristic creates. This section explains how a heuristic constructs a solution, which is then assessed to obtain the fitness of the heuristic. The heuristics consist of functions and terminals explained in section 4.3.2. Section 4.3.3 gives a detailed example of the packing process to further clarify how a heuristic is applied.

To pack all of the pieces in a problem instance, the heuristic operates within a fixed framework, shown in algorithm 3. For the first bin, the heuristic is evaluated once, and therefore returns a numerical value for the bin (this process is explained in detail in section 4.3.3). The numerical value is treated as the 'score' for the bin, which is stored as the best score so far. At the next bin, the heuristic is evaluated again, and the score for this bin is compared to the best score so far. If it is larger, then the *current* bin is stored as the best so far and its score is stored in memory. After all of the bins have been given a score by the heuristic, the piece is placed into the bin which received the highest score.

This framework differs fundamentally from the framework of chapter 3, where the piece was put into the current bin immediately if the heuristic returned a value greater than zero. The framework used in this chapter will require more evaluations of the heuristic, because *all* of the bins are evaluated for every piece. However, this will also provide the heuristic with more choice and, therefore, the heuristics evolved have more potential to

create better solutions.

In summary, we evolve a heuristic which scores each bin for its suitability for the current piece. To construct a solution, the framework shown in algorithm 3 iterates through all of the bins, applying the heuristic to each one, and therefore obtaining a numerical value for each. The bin which receives the highest value is the bin into which the piece is placed.

Algorithm 3 Pseudo code of the framework within which the heuristics are applied in chapter 4. The inputs are a problem instance set I, each $i \in I$ contains a list L of pieces to be packed

```
1: for each problem i ∈ I do
2:    initialise partial solution s with one empty bin
3:    for each piece p in L do
4:       variable bestSoFar = −∞
5:       for each bin b in s do
6:          heuristic_output = evaluateHeuristic(bin b)
7:          if heuristic_output > bestSoFar then
8:             bestSoFar = output
9:             bestBinIndex = b
10:         end if
11:      end for
12:      put piece in bestBinIndex bin
13:      if piece was placed in the empty bin then
14:         add a new empty bin to s
15:      end if
16:   end for
17:   fitness_of_heuristic += fitness of solution to s
18: end for
```

The heuristic is not constrained by the framework to only consider the bins which have enough space for the current piece. As was the case in chapter 3, the heuristic is permitted to overfill a bin if it chooses to. Such heuristics are penalised heavily by the fitness function, however, and so are highly unlikely to be chosen as parents of the individuals in the next generation. The result is that the evolved heuristics always contain some mechanism

which ensures that a bin is never overfilled. However, in contrast to the heuristics evolved in chapter 3, there is no possibility of the heuristic *not* putting a piece into any bin. The first bin is always stored as the best so far, so if no other bin receives a higher score then the piece will be placed there.

4.3.2 Structure of the Heuristics

The heuristics evolved by the genetic programming hyper-heuristic have a tree structure, consisting of function and terminal nodes. The functions and terminals used in the experiments in this chapter are given in table 4.1. For a full explanation of the role of functions and terminals, refer back to section 2.7.1.

For a heuristic to produce a solution, it considers each piece in sequence, and decides which bin to pack the current piece into. It chooses a bin by operating within a fixed framework (shown in algorithm 3 and explained in section 4.3.1), which evaluates the heuristic once for each bin. When the heuristic is evaluated, its terminal nodes each acquire a value, dynamically determined by the properties of the current bin and the current piece. The heuristic returns one numerical result, by evaluating each of its function nodes using the values supplied by its terminal nodes (this is explained in more detail in section 4.3.3).

The function set includes the four basic arithmetic operators, and a 'less-than-or-equal-to' operator. The 'absolute value' function, which was used in chapter 3 because it is a standard function, is not used here because it could in fact be counter-productive and we wish to investigate if good quality heuristics can be evolved without it. For example, it is important to know whether the result of evaluating a heuristic is positive or negative. It can mean the difference between interpreting the bin as a very good candidate or very bad. Calculations that would distinguish between the two extremes could be rendered obsolete by the absolute value terminal. We do not wish the genetic programming to waste effort when the rest of the functions are sufficient. There are three terminals, 'F', 'S', and 'C'. F represents the total size of the pieces already in the bin, C represents the capacity of the bin, and S represents the size of the current piece.

The heuristic must evolve to combine information about both the piece and the bin, in order to ensure the constraints of the bin packing problem are adhered to. This is achieved by using the function nodes to perform operations on the values of the terminal nodes and the values returned by other function nodes.

TABLE 4.1: The functions and terminals used in chapter 4, and descriptions of the values they return

	Symbol	Arguments	Description
Functions	$+$	2	Add
	$-$	2	Subtract
	$*$	2	Multiply
	$\%$	2	Protected divide function. A division by zero will return 1
	$\leq$	2	Tests if the first argument is less than or equal to the second argument. Returns 1 if true, -1 if false
Terminals	F	0	Returns the sum of the pieces already in the bin
	C	0	Returns the bin capacity
	S	0	Returns the size of the current piece

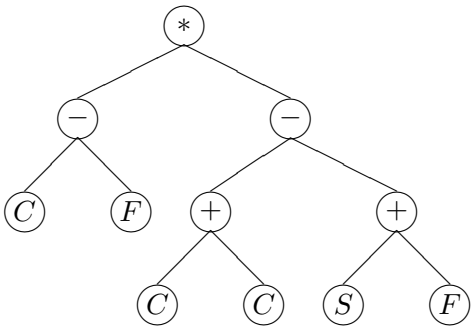

FIGURE 4.5: An example heuristic, used in the example packing described in section 4.3.3 and figures 4.6 to 4.9

4.3.3 Simple Example of a Heuristic Constructing a Solution

This section works through a simple example of a heuristic packing three pieces from an example problem instance. This is to further clarify the methodology explained in section 4.3.1. Figure 4.5 shows the heuristic that will be used in the example, and the instance to be packed is shown in figure 4.6. This instance is identical to the instance used to explain how the best-fit heuristic operates (section 4.2), and so it may be useful to refer back to that section for a comparison.

Figure 4.6 shows that the first piece to pack has a size of 70. The heuristic is evaluated at each bin, and obtains a different score for each because the terminal nodes acquire different values at each. In expression 4.2, the heuristic is written in linear form, rather than tree form. When the heuristic is evaluated at the first bin, the C terminal will

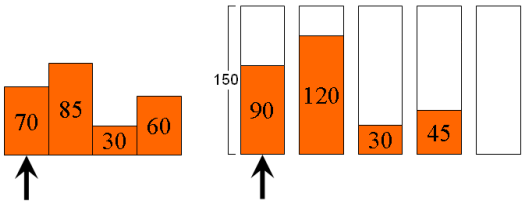

FIGURE 4.6: The initial partial solution from section 4.3.3. The heuristic in figure 4.5 will pack the first three pieces.

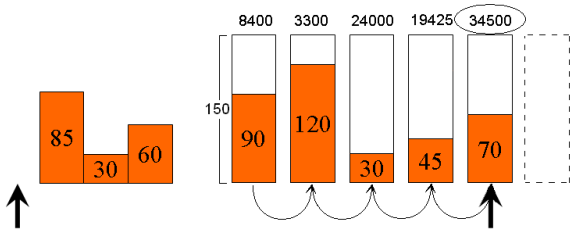

FIGURE 4.7: All of the bins are scored by the heuristic, the bin on the far right scores the highest with 34500. A new empty bin is opened because the existing one receives the piece.

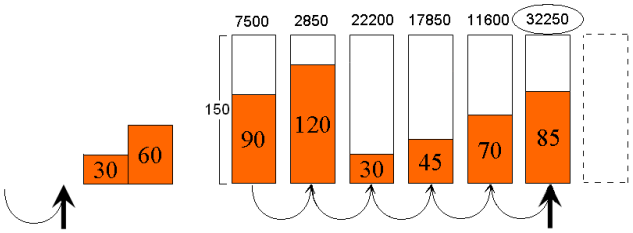

FIGURE 4.8: The newly opened bin on the far right scores the highest with 32250. Again, a new empty bin is opened because there must always be one available.

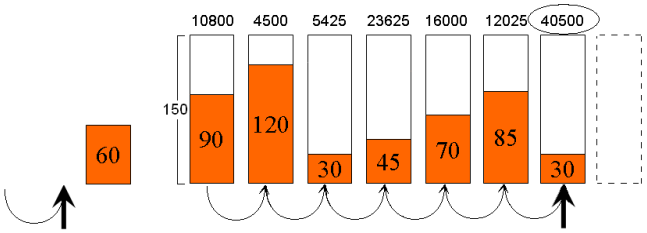

FIGURE 4.9: For the third time, the empty bin receives the piece. The heuristic appears to prefer to place each piece into a new bin.

acquire a value of 150, the S terminal will acquire a value of 70, and the F terminal will acquire a value of 90. Expression 4.1 shows the heuristic with those values substituted in. This expression evaluates to 8400, so this is the score for the first bin.

$$(C - F) * ((C + C) - S - F) \tag{4.1}$$

$$(150 - 90) * ((150 + 150) - 70 - 90) \tag{4.2}$$

Figure 4.7 shows the scores that are given to the four other bins. It also shows that the piece is put into the fifth bin, because this obtains the highest score. A new bin is opened after the piece is placed into the fifth bin. There is always an empty bin available to the heuristic.

The next piece has a size of 85, and the process of packing it is shown in figure 4.8. The bin that was opened after the previous piece was placed (drawn as a dotted line in figure 4.7) is now available, so there are now six bins for the heuristic to choose from. Of the six bins, the empty bin on the far right receives the highest score, 32250, so the piece is placed into that bin. A new bin is opened again, because the piece was placed into the empty bin. Figure 4.9 shows the next piece being packed into the newly opened seventh bin.

This heuristic appears to select the empty bin for each new piece. This is a strategy that will not result in a good solution, but the solution will at least be a legal one because a bin will never be overfilled.

4.3.4 How the Heuristics are Evolved

Section 4.3.1 has explained how a heuristic is applied to a problem instance to obtain a solution, and section 4.3.2 has described each of the potential components of a heuristic. This section explains how the heuristics themselves are evolved by the genetic programming algorithm. There are three parts to this section. The first part will provide an overview of the genetic programming parameters. The second part will describe the data sets on which we evolve the heuristics. The third part will describe the fitness function used to assess the performance of each heuristic.

TABLE 4.2: Genetic programming initialisation parameters for chapter 4

Population Size	1000
Generations	50
Maximum Depth of Initial Trees	4
Crossover Proportion	90%
Reproduction Proportion	10%
Selection Method	Roulette wheel

Parameters

The experiments in this chapter use the genetic programming parameters shown in table 4.2. As stated in chapter 1 on table 1.1, the genetic programming system used for the experiments of this chapter is the author's own code. Subsequent chapters use the ECJ (Evolutionary Computation in Java) package (http://www.cs.gmu.edu/∼eclab/projects/ecj/).

The population size is set to 1000, and the number of generations is set to 50. These are the parameters with which the genetic programming performed well in the experiments of chapter 3. The maximum depth of the initial randomly created trees is four. In contrast to chapter 3, we apply a parsimony pressure on the population to control the effects of code bloat. Any individual 30 nodes larger than the average of the population is penalised, by assigning it the same fitness given to a heuristic that produces an illegal solution. This is similar to the 'Tarpeian wrapper' method [226].

The proportion of the population created from the crossover and reproduction operators is 90% and 10% respectively, and individuals are selected for these operators with fitness proportional (roulette wheel) selection.

The Data Sets

The experiments presented in this chapter are based around seven problem classes, each defined by a uniform piece size distribution. The classes are referred to as C_{l-u} where l is the lower limit of the distribution and u is the upper limit.

The piece size distributions used in the randomly generated instances are shown graphically in figure 4.10. From this figure, one can see that the seven classes are arranged into three levels of a hierarchy. We refer to C_{10-49} and C_{50-89} as sub-problems of C_{10-89}. In the same way, C_{10-29} and C_{30-49} are sub-problems of C_{10-49}, and C_{50-69} and C_{70-89} are sub-problems of C_{50-89}. This is because, as can be seen in Fig. 4.10, all of the values

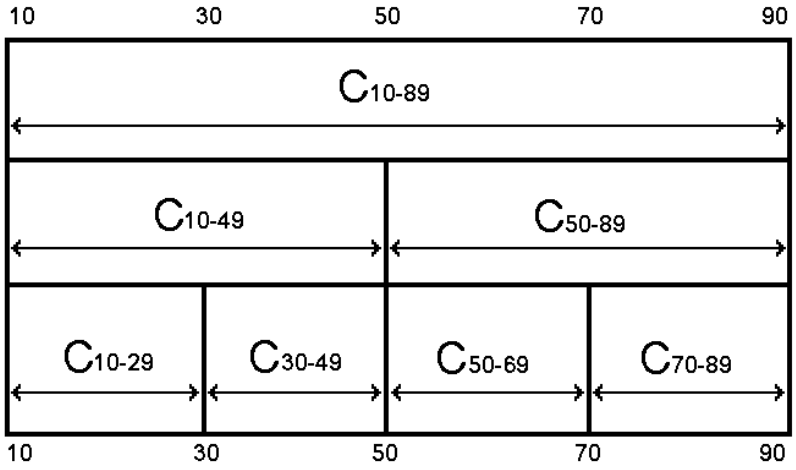

FIGURE 4.10: Diagram of the piece size distributions of each class of instances created for the experiments of chapter 4

in a sub-problem can appear in its super-problem, but not all of the values that appear in the super-problem can appear in the sub-problem. So the sub-problems are less general as the pieces have a smaller range of values.

From each of the seven classes, we generate two unique 'sets' of twenty problem instances, a 'training' set and a 'validation' set. Each instance has 120 pieces uniformly distributed between the two limits defined by the class. The bin capacity is always 150. The defining characteristic of a set is therefore the uniform distribution that its piece sizes are taken from. Training sets and validation sets are referred to as T_{l-u} and V_{l-u} respectively, where l is the lower limit of the uniform distribution and u is the upper limit.

A heuristic is always evolved using a particular training set of twenty instances. The validation sets are used to test the heuristics on new problems after they have been evolved, and as such they are not involved in the evolution stage. We evolve 50 heuristics for each class, which are obtained from 50 runs of genetic programming. One heuristic is taken from each run as the best heuristic found. Each set of 50 heuristics is defined as H_{l-u}, where l and u are the lower and upper limits of the uniform distribution of the training set they were evolved on. There are 350 heuristics in total.

In the training stage, a heuristic will only come into contact with one training set. In the validation stage, the heuristic is separately tested on all seven validation sets. This will enable us to investigate how re-usable the heuristic is on new problems of the same class, and new problems of different classes.

The Fitness Function

The fitness function employed in the experiments of this chapter is shown in equation 4.3, where n_j = number of bins used in instance j, $fullness_{ij}$ = sum of all the pieces in bin i of instance j, and C = bin capacity. It is identical to the second fitness function used in chapter 3 except that its formulation here highlights the fact that the total of 20 instances is always taken. It puts a premium on bins that are filled completely or nearly so. Importantly, it avoids the potential problem of plateaus in the search space, that occur when the fitness function is simply the number of bins used by the heuristic [97]. Smaller fitness values are better.

$$Fitness = \sum_{j=1}^{20} \left(1 - \left(\frac{\sum_{i=1}^{n_j} (fullness_{ij}/C)^2}{n_j} \right) \right) \tag{4.3}$$

A fitness of 1000 is given to any heuristic which produces an illegal solution, an arbitrarily high value compared to the range of values that the fitness function can return. In contrast to chapter 3, a parsimony pressure is employed to control code bloat, so any heuristic at least 30 nodes larger than the average of the population also receives a fitness of 1000.

4.4 Results

This section presents the results of the experiments performed using the methodology explained in this chapter. Section 4.4.1 presents an example of an evolved heuristic, to give the reader an impression of the variety of behaviours that the heuristics can evolve, especially their behaviour on the validation sets of classes other than the one they were evolved on. The next three sections present three aspects of the results of all of the 350 evolved heuristics. Section 4.4.2 analyses the quality of the evolved heuristics. Section 4.4.3 analyses to what degree the heuristics can be specialised for sub-classes with progressively narrower distributions. Finally, section 4.4.4 analyses the robustness of the heuristics on new problems, specifically if the heuristics actually produce illegal results on classes different to that which they were evolved on.

4.4.1 Example of an Evolved Heuristic

This section presents and explains a 35 node individual which was evolved on T_{10-90}. This heuristic is a good example of how a heuristic can perform well on its training set, and the validation set from the same class, but badly on validation sets from other classes.

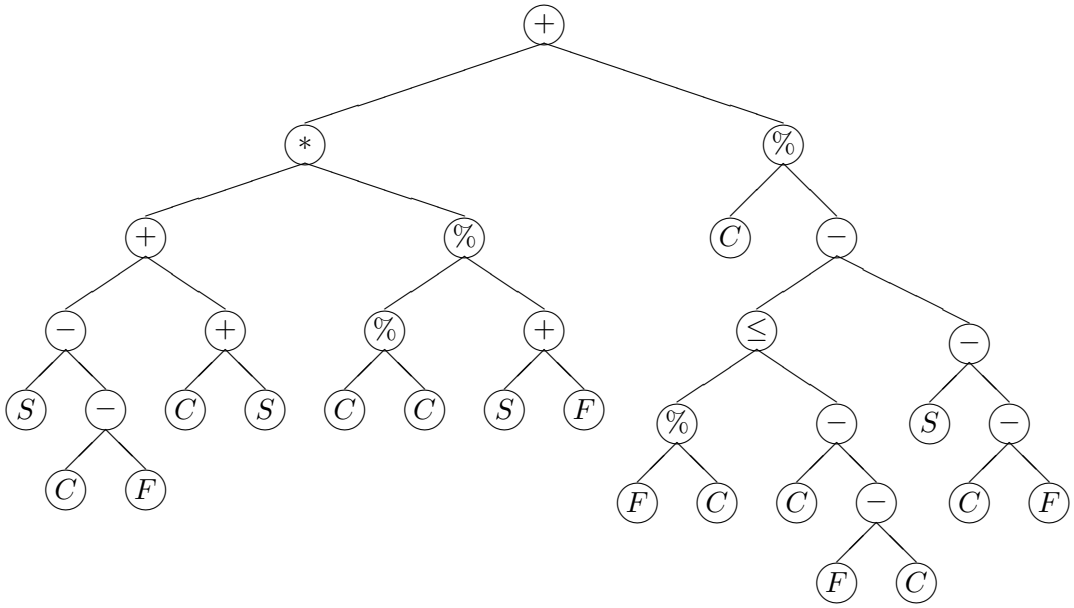

FIGURE 4.11: An example heuristic, evolved on T_{10-90}

Figure 4.11 shows the evolved heuristic in full, and equation 4.4 shows the heuristic represented as a mathematical equation. Table 4.3 shows the results that this heuristic obtains on the seven validation sets. The table shows that the heuristic uses at most one more bin than best fit on each validation set, apart from sets V_{10-29} and V_{10-49}.

$$\frac{2S+F}{S+F} + \frac{C}{((\frac{F}{C}) \leq (2C-F)) + (C-S-F)} \tag{4.4}$$

Indeed, the result for V_{10-29} represents the heuristic putting each piece in a new bin. This can be explained as follows, with reference to equation 4.4. Given a piece size between 10 and 29, no fullness between 10 and 29 can make the heuristic evaluate to a result greater than it does when the fullness is zero. So, the score that the heuristic gives to the non-empty bins will always be lower than the score that it gives to the empty bin. Therefore, no piece will ever be put in a bin which already contains a piece, it will always be placed in an empty bin.

However, in all other classes, where the pieces can be larger, there are combinations of piece sizes and fullness that can make the heuristic return a value *greater* than when the fullness is zero. So it is possible for pieces to be put in bins that already contain a piece.

TABLE 4.3: Summary of the results obtained by the heuristic shown in figure 4.11 compared to the results of best-fit. The result shown is the total bins used over the 20 instances.

Validation set	Heuristic result	Best-fit result
V_{10-89}	835	834
V_{10-49}	676	503
V_{50-89}	1220	1220
V_{10-29}	2400	326
V_{30-49}	702	704
V_{50-69}	1198	1198
V_{70-89}	1815	1815

But still, no piece with a size of less than 30 will ever be put in a bin with a fullness of less than 30. The heuristic's result on V_{10-49} also shows this difficulty that the heuristic has when packing small pieces. However, the result is better than for V_{10-29} because some of the pieces are large enough for the fullness to reach a high enough value that more pieces can be put in the same bin.

It is worth noting that, while the result for V_{10-29} is very poor, none of the solutions are illegal. The safeguard against illegal solutions is the right hand term of the two terms in equation 4.4, and it is a safeguard that holds in all problem classes. If putting a piece in a bin would overfill the bin by one unit, then the right hand term becomes equal to 1 (a division by zero returns 1), which is low enough for the whole heuristic to value an empty bin more. Therefore, the heuristic will never choose a bin that would be overfilled over a bin that the piece can fit into. If the bin would be overfilled by two or more units, then the heuristic's result becomes negative, and again the heuristic will rate the open bin more highly than it rates an illegal placement.

Equation 4.4 is an example of a heuristic strategy that uses the ratios of piece sizes very well, bin fullness and bin capacity that exist in the class it was evolved on, but cannot exist in certain other classes.

4.4.2 The Quality of Evolved Heuristics

In this section of results, we are concerned with the performance of the heuristics on the validation set of the class they were evolved on. This shows their quality on new unseen problem instances. Recall that 50 heuristics are evolved for each class, making 350 in total, and each heuristic is the result of one run of genetic programming. We compare the performance of all 350 heuristics with the best-fit heuristic, on the validation set of the class

TABLE 4.4: The performance of each set of evolved heuristics tested against best-fit

Validation Set	Better than Best-Fit	No Significant Difference	Worse than Best-Fit
H_{10-89}	0	48	2
H_{10-49}	0	50	0
H_{50-89}	0	49	1
H_{10-29}	5	45	0
H_{30-49}	6	44	0
H_{50-69}	0	29	21
H_{70-89}	3	47	0

that they were evolved on. A two-tailed paired-difference t-test, with a confidence level of 95%, is used to test for a significant difference from best-fit (this test is explained in section 3.4.2).

Table 4.4 summarises the results of the comparisons with best-fit. The table shows, for each evolved heuristic set, the number of heuristics that are significantly better, and worse, than best-fit. The vast majority of heuristics show no significant difference with best-fit. However, there are 14 evolved heuristics that perform significantly better than best-fit on their validation sets. These are all produced from training on classes at the bottom of the hierarchy. In other words they are produced on the narrowest problem distributions. From this, we can conclude that it is more probable for a heuristic to become specialised in these classes due to the piece size distribution being sufficiently narrow.

The set H_{50-69} is clearly an exception, where 21 heuristics perform significantly worse than best-fit on V_{50-69}, and none perform better than best-fit. This anomalous result can be attributed to the particularly good performance of best-fit on this class of piece sizes. It would seem that best-fit performs well when the pieces are all roughly between 1/3 and 1/2 of the bin capacity, and so finding a heuristic that outperforms best-fit is more difficult in this class.

4.4.3 Specialisation of Evolved Heuristics

In this section, we analyse how heuristic sets perform on the validation set from the class upon which they were evolved, compared to the heuristic sets evolved on the super-class and super-super-class of that validation set. For example, the super-class of P_{10-29} is P_{10-49}, and its super-super-class is P_{10-89}. The super-class of P_{50-89} is P_{10-89}, and has no super-

TABLE 4.5: A comparison of the performance of six sets of evolved heuristics with the heuristic sets evolved on their respective super-class and super-super-class. The entries in the table state whether the set is better, worse, or the same as its super-class (or super-super-class) set. The values in brackets represent the probabilities that the performance values of both heuristic sets come from underlying populations with the same mean. For example, row 4 (labelled H_{10-29}) shows that the H_{10-29} set is better on the V_{10-29} instances than the set evolved on its super-class (H_{10-49}) and the set evolved on its super-super-class (H_{10-89})

Heuristic Set	Super-Class Set	Super-Super-Class Set
H_{10-49}	Better (≤ 0.01)	N/A
H_{50-89}	Same (0.35)	N/A
H_{10-29}	Better (≤ 0.01)	Better (≤ 0.01)
H_{30-49}	Same (0.29)	Better (≤ 0.01)
H_{50-69}	Same (0.09)	Worse (≤ 0.01)
H_{70-89}	Better (≤ 0.01)	Better (≤ 0.01)

super-class (indicated by N/A in table 4.5). There are 50 fitness scores for each heuristic set. This set of 50 fitness scores is compared to the 50 from the other heuristic set, to assess if there is a significant difference between the two sets.

The results shown in table 4.5 are the results of a one-tailed t-test between the set of 50 total scores of one heuristic set and the 50 total scores of another. In each comparison, we are testing the hypothesis (H_1) that the values of the sub-class come from a distribution with a mean lower (better) than the distribution of the super-class. The null hypothesis (H_0) is that the values of both the sub-class and the super-class come from distributions with the same mean. The values shown represent the probability that H_0 is true. We use a 95% significance level, so if the probablility is lower than 0.05 then the results of the heuristic set from the sub-class are significantly different.

Table 4.5 shows that, as the classes become more specialised, the heuristics evolved on them can be more specialised too. This hierarchy is observed in all situations except for one anomaly, where set H_{10-89} is significantly better at V_{50-69} than H_{50-69}. In contrast to the other results in table 4.5, we use a two-tailed t-test for this comparison of H_{50-69} with its super-super-class. This is because the mean of H_{50-69} is higher (worse) than the mean of its super-super-class, and therefore we test whether the distributions are significantly different in either direction, as opposed to testing H_1.

There exists an epithet *"Jack of all trades, master of none, though ofttimes better than master of one"*. The distribution of the piece sizes of V_{50-69} and their ratio with the

capacity of the bins evidently present a situation where the jack of all trades is better than the master of one. The heuristics of H_{10-89}, which represent the jack of all trades in this case, are able to use their wider skills to perform better on average than heuristics which are specialised in class C_{50-69}.

There is no statistically significant difference between the mean performances of H_{50-89}, H_{50-69} and H_{30-49} on their native classes compared to the performances of heuristics evolved on their super-classes. There seems to be no specific features of these classes that heuristics can learn through evolution that they cannot already learn by packing the pieces of their super-class. In the experiments, it appears that piece sizes of around 1/3 of the bin capacity apparently inhibit the specialisation of heuristics on these classes. Conversely, there appears to be more scope for specialisation at the extremes of the range of piece sizes used here. Specifically, H_{10-29} and H_{70-89} are the only sets of heuristics with means that are significantly better than their super-class and their super-super-class at a 98% level of confidence.

4.4.4 Robustness of Evolved Heuristics

The robustness of a heuristic is important in a commercial scenario. Where there are financial considerations, practitioners may be reluctant to trust an evolved heuristic to perform well on new problems that it is given. Whilst we are not suggesting that the heuristics evolved here could be employed commercially, possible mistrust of computer generated heuristics is an important issue that future work on automatic heuristic generation will have to address.

All the evolved heuristic sets were tested on all seven of the validation sets, as explained in section 4.3.4, but table 4.6 presents only the results where the heuristic set is tested on an 'unrelated' validation set, containing piece sizes that the heuristics did not pack during their evolution. Each class, apart from C_{10-89}, has one unrelated class in the middle layer of the hierarchy and at least two unrelated classes in the lower level of the hierarchy. For example, class C_{10-49} has three unrelated classes, which are C_{50-89} in the middle of the hierarchy, and C_{50-69} and C_{70-89} in the bottom row of the hierarchy. In table 4.6, a cross ($\times$) represents at least one illegal result on the validation set by the 50 heuristics in the set. A circle ($\circ$) represents no illegal results by that heuristic set on the validation set.

TABLE 4.6: Summary of the illegal results when heuristic sets are each tested on three unrelated validation sets, containing piece sizes they did not pack in their evolution. A cross represents at least one illegal result, a circle represents no illegal results

	Validation sets of three classes unrelated to the heuristic set		
Heuristic Set	Mid-level	Two low-level classes	
H_{10-49}	×	×	×
H_{10-29}	×	×	×
H_{30-49}	×	×	×
H_{50-89}	×	×	○
H_{50-69}	○	○	○
H_{70-89}	○	○	○

H_{50-89} produces some illegal results on V_{10-29} but not on its subclasses V_{50-69} and V_{70-89}. In the same way, H_{10-49} produces some illegal results on V_{50-69} and V_{70-89} but not on its subclasses V_{10-29} and V_{30-49}. This is summarised in the first and fourth rows of table 4.6. In addition to this (but not shown in the table), the most general set of heuristics H_{10-89}, produces no illegal results over any of the validation sets. These results show that heuristics evolved on a super-class can be applied to sub-classes without them failing.

However, when applied to an unrelated class, there is a chance that the heuristic will not produce a legal solution. This can be explained by the fact that the heuristics, when applied to an unrelated class, will be trying to pack piece sizes that they have not been evolved to pack, and there is a chance that a heuristic may try to use relationships of piece sizes, fullness, and capacity that cannot exist in the new class.

To help explain the next point, let set A be the group of classes C_{10-49}, C_{10-29} and C_{30-49}. Also, let set B be the group of classes C_{50-89}, C_{50-69} and C_{70-89}. Table 4.6 shows that set A seems to be easier for the heuristics trained on set B, than set B is for the heuristics trained on set A. This can be seen by comparing the top three rows of table 4.6 to the bottom three rows. For example, on set A, only two heuristics from H_{50-69} result in some illegal solutions, and they only result in illegal solutions on V_{10-49} and V_{10-29}. In contrast to this, H_{10-49}, H_{10-29}, and H_{30-49} all result in some illegal solutions on each of V_{50-89}, V_{50-69} and V_{70-89}.

Our results show that the heuristics evolved for the distributions above 1/3 of the bin capacity can mostly be applied to lower piece sizes (piece sizes of less than 50) without producing illegal solutions, even if these heuristics do not result in equivalent performance in the lower distributions. The reverse is true for the bin packing heuristics evolved for

lower piece size distributions. These heuristics are less applicable to problems with higher piece sizes, and result in more illegal solutions.

To summarise the robustness results, they show how important it is that the training set used to evolve a heuristic is representative of the future problems that the heuristic is expected to encounter. The results show that if a heuristic is presented with pieces it has not seen before, then there is a chance it will not be able to function as intended and illegal solutions may result. However, the results also show that no heuristic generated automatically by our genetic programming methodology produced an illegal result when applied to new problems of the same class.

4.5 Conclusion

There are three major conclusions to be drawn from the experiments performed in this chapter. Firstly, the vast majority of the 350 automatically designed heuristics evolved in this chapter match the performance of the human-created best-fit heuristic. In the previous chapter, no heuristics of this category were evolved over 20 instances. This increase in performance of the evolved heuristics is due to the change in the way that they are applied to the problem, allowing them to consider all of the bins before making a decision. The methodology of the previous chapter did not afford the heuristics this choice.

Secondly, heuristics can be evolved to be specialists on a particular sub-problem, or general enough to work on all sub-problems. However, there is a trade-off between performance and generalisation. Three levels of generalisation were investigated, creating a hierarchy structure of piece size distributions. In general, the heuristics evolved on narrower distributions performed better, on new instances of that distribution, than heuristics which were evolved to be more widely applicable.

The third main conclusion is that heuristics evolved on a super-class will not produce illegal results on problems from one of its sub-classes. However, there is a chance that if a heuristic is applied to problems of a different class then illegal solutions may be produced. This means that the hyper-heuristic creates heuristics which are robust when packing pieces they have seen before during their evolution.

These three conclusions show, above all, that care must be taken to ensure that a representative training set is provided when automatically evolving heuristics. The heuristics generated by this type of system will all be specialised to some degree (for example,

even the most general heuristics evolved here are 'specialised' to the distribution 10-89), and just because a heuristic can solve a bin packing problem, it does not mean it can be relied upon to obtain a solution for *all* bin packing problems.

In contrast to the conventional hyper-heuristic approaches and other metaheuristics, our genetic programming hyper-heuristic approach has two outputs, good quality solutions, and a heuristic that can be reused on future problems of the same type without further training. The additional output of a re-usable heuristic means that the computationally expensive evolution process can be employed to produce a heuristic on small problems, and then the resulting heuristic can be applied quickly to larger problem instances. In this way, the solutions to the larger instances could potentially be obtained much more efficiently than if an evolutionary algorithm is employed on the larger instances directly.

This re-usability of the heuristics is an important quality of the hyper-heuristic approach. Therefore, in the next chapter, we will investigate how the heuristics perform when applied to instances that are orders of magnitude larger than their training instances. The results will show whether the evolved heuristics maintain their performance when they pack a greater number of pieces than were in their training set, and what effect changing the size of the training set has.

Chapter 5

The Scalability of Evolved Online Bin Packing Heuristics

5.1 Introduction

Chapter 4 showed that a genetic programming hyper-heuristic can generate heuristics which perform as well as the human designed best-fit heuristic on unseen problem instances. Seven problem classes were created, each defined by the upper and lower limits of a uniform distribution. The experiments investigated the quality, specialisation, and robustness of the evolved heuristics.

Chapter 4 highlighted a benefit of the hyper-heuristic approach over traditional evolutionary computation methods which operate on a space of solutions. If there are 1000 problems to be solved, then it is possible to evolve a heuristic on a representative sample of 20 instances. Assuming that the heuristic is constructive, it can then be applied to the remaining 980 instances very quickly. This can be compared to a traditional genetic algorithm which would have to be run on each instance (i.e. 1000 separate runs). Assuming that a genetic algorithm takes longer than a constructive heuristic, the hyper-heuristic methodology would be quicker. The evolution of a heuristic may require more time than a single genetic algorithm run over 20 instances, but once the heuristic is evolved, it is very quick to obtain a solution for the remaining instances, whereas the genetic algorithm would have to be run again for each instance.

This chapter investigates a potential further benefit of the hyper-heuristic methodology. Utilising traditional evolutionary computation methods is computationally expensive. Larger problem instances require more time to obtain a solution, and it is often difficult

to devise an efficient representation of the problem for an evolutionary algorithm to operate on.

The experiments of this chapter show that a heuristic can be evolved on a small instance, and then applied to solve larger instances. Assuming that the evolved heuristic is quick to run, and constructive heuristics generally are, then this methodology has the potential to greatly reduce the computational cost of solving larger instances. Again, the benefit is possible because the output of this hyper-heuristic methodology is a re-usable heuristic, as well as a solution to the problem instance on which it was evolved.

The experiments performed in this chapter involve 20 training instances each with 500 pieces, and 20 validation instances each with 100,000 pieces. The heuristics are evolved on the smaller instances and then their performance on the unseen larger instances is recorded. Section 5.2 presents the framework within which the heuristics are applied, the structure of the heuristics, and the genetic programming parameters. Section 5.3 discusses the results, over the full 100,000 pieces and in detail over the the first 500 pieces. The results compare the effect of evolving the heuristics on all 500 pieces of the training set, or only the first 100 or 250 pieces. Section 5.4 presents the conclusions to this chapter.

5.2 Methodology

5.2.1 How the Heuristic is Applied

The heuristics are applied in the same way as in chapter 4, where an example of the packing process is given in section 4.3.3. A heuristic operates within a fixed framework, shown in algorithm 4, which applies the heuristic to pack all of the pieces in the instance. Algorithm 4 is identical to algorithm 3 in the previous chapter, and is repeated here as a new algorithm for ease of reference.

The pieces are packed one at a time in the order that they appear in the list, this is shown on line 2 of the algorithm. To decide where to pack a piece, the heuristic is evaluated on each bin in sequence. This process is shown on lines 4 and 5 of the algorithm. At the first bin, the heuristic is evaluated once, and therefore returns a numerical value for the bin. The numerical value is treated as the 'score' for the bin, which is stored as the best so far. At the next bin, the heuristic is evaluated once again, and the resulting score for this bin is compared to the best score so far (see line 6). If it is larger, then the *current* bin is stored as the best so far and its score is stored in memory on line 7. After all of the

Algorithm 4 Pseudo code of the framework within which the heuristics are applied in the experiments of chapter 5. This is identical to algorithm 3 in the previous chapter, as the way the heuristics are applied is kept the same. The inputs are a problem instance set I, each $i \in I$ contains a list L of pieces to be packed

```
1:  for each problem i ∈ I do
2:    initialise partial solution s with one empty bin
3:    for each piece p in L do
4:      variable bestSoFar = −∞
5:      for each bin b in s do
6:        heuristic_output = evaluateHeuristic(bin b)
7:        if heuristic_output > bestSoFar then
8:          bestSoFar = output
9:          bestBinIndex = b
10:       end if
11:     end for
12:     put piece in bestBinIndex bin
13:     if piece was placed in the empty bin then
14:       add a new empty bin to s
15:     end if
16:   end for
17:   fitness_of_heuristic += fitness of solution to s
18: end for
```

bins have been given a score by the heuristic, the algorithm progresses to line 11, where the piece is placed into the bin which received the highest score. Line 13 shows that the fitness of the heuristic is the sum of the quality of the solutions it creates to the 20 instances.

In summary, we evolve a heuristic which scores each bin for its suitability for the current piece. To construct a solution, the framework shown in algorithm 4 iterates through all of the bins, applying the heuristic to every one in turn, and therefore obtaining a numerical value for each bin. The bin which receives the highest value is the bin into which the piece is placed. The heuristics' performance is judged by the quality of their solutions to 20 instances, the details of which are given in section 5.2.3.

The heuristic is not constrained by the framework to only consider the bins which

TABLE 5.1: The functions and terminals used in chapter 5, and descriptions of the values they return

	Symbol	Arguments	Description
Functions	$+$	2	Add
	$-$	2	Subtract
	$*$	2	Multiply
	%	2	Protected divide function. A denominator of zero will be changed to 0.5
Terminals	E	0	Returns the emptiness of the bin, the sum of the pieces in the bin subtracted from the capacity of the bin
	S	0	Returns the size of the current piece

have enough space for the current piece. As was the case in chapters 3 and 4, the heuristic is permitted to overfill a bin if it chooses to. However, such heuristics are heavily penalised by the fitness function, and so are highly unlikely to be chosen as parents of the individuals in the next generation. The result is that the evolved heuristics always contain some mechanism which ensures that a bin is never overfilled.

5.2.2 Structure of the Heuristics

The heuristics evolved by the genetic programming hyper-heuristic have a tree structure, consisting of function and terminal nodes. The functions and terminals used in the experiments in this chapter are given in table 5.1. For a full explanation of the role of functions and terminals, refer back to section 2.7.1. The functions and terminals that are used in the experiments in this chapter are similar to those of chapter 4, but not identical. The differences are explained in this section.

The functions are shown in the first four rows of table 5.1. These are the same functions that were used in the previous chapters. However, in contrast to the function set of chapter 4, the '$\leq$' function has been removed. The $\leq$ function existed to provide an easy way for the heuristic to differentiate between bins that a piece can fit into and those that the piece cannot. It was included in previous function sets because intuitively it seems useful, and it is a function that would be used by a human programmer when creating the first-fit heuristic. However, a heuristic can be built that performs the same function, without the $\leq$ function. For example, simply subtracting the bin fullness and the piece size from the bin capacity will give a negative result if the bin is overfilled, and a positive one if not, so

TABLE 5.2: The values returned by the standard protected divide function and the modified protected divide used in the experiments of this chapter.

Calculation	$\frac{10}{1000}$	$\frac{10}{100}$	$\frac{10}{10}$	$\frac{10}{1}$	$\frac{10}{0}$
When % returns 1	0.01	0.1	1	10	1
When denominator is changed to 0.5	0.01	0.1	1	10	20

the $\leq$ function may not be necessary for the function set to be sufficient.

The 'protected divide' function is slightly different to its function in the previous two chapters. The function exists so that a division by zero will return a value rather than an error. Previously, the function returned a value of one if it was asked to divide by zero. This is modified slightly for the experiments in this chapter. It now changes the denominator to 0.5, which has the effect of doubling the numerator.

The logic behind this decision can be seen in table 5.2. The top row of the table shows five fractions with denominators descending from 1000 to 0 in powers of ten. The middle row shows the result of the protected divide function as implemented in the first two chapters, where a division by zero returns a value of one. The values returned by this function get progressively higher until there is a division by zero in the last column, where one is returned. Compare this to the lower row of table 5.2, which shows the values returned by the protected divide function that changes a denominator of zero to 0.5. This time, the values get progressively higher even in the last column.

It makes logical sense that the function should not return one when the denominator is zero, otherwise the results are identical when the denominator is either ten or zero. For example, if the denominator represents the space left in the bin, then it would not distinguish between a completely full bin and one that has ten free units of space, and everything between these two values would be rated higher than both.

The terminals are shown in the lower two rows of table 5.1. In the previous two chapters, three terminals have been used, F, C, and S. A new terminal (E) is defined as $C - F$, or the capacity minus the fullness of the bin, which is the 'emptiness' of the bin.

Experiments were performed with both sets of terminals to determine if there is a benefit in reducing the size of the set from three to two. Reducing the terminal set does not result in better evolved heuristics. However, heuristics of the same quality *are* found in less generations. We conclude that this is because the calculations the heuristics perform at each bin with the C and F terminals are mostly of the form $C - F$, and replacing these two

nodes with one node (E) removes the effort needed to evolve this structure. A potential disadvantage of removing C is that there is now no terminal that represents a constant. While constants are sometimes important in genetic programming, the experiments using both sets of terminals suggest that a terminal with a constant value is not important for this problem.

5.2.3 How the Heuristics are Evolved

This section explains the parameters used by the genetic programming algorithm to evolve the heuristics. There are three parts to this section. The first part will provide an overview of the genetic programming parameters. The second part will describe the data sets on which we evolve the heuristics. The third part will describe the fitness function used to assess the performance of each heuristic.

Parameters

For the experiments in this chapter, the genetic programming system is implemented using the ECJ package, a Java based evolutionary computation research system, which can be found at http://www.cs.gmu.edu/∼eclab/projects/ecj/. Many of the default parameters of ECJ are the same, or similar, to those selected for the experiments in the two previous chapters, and so they are left unchanged. The reason for this change from the author's code, which was used in the previous chapters, is because ECJ is a proven system with a relatively large community that supports it. The ECJ package was found by the author to be faster, and more flexible.

The population size is set to 1024, and the number of generations is set to 50. The proportion of the population created from the crossover and reproduction operators is 90% and 10% respectively, and individuals are selected for these operators with tournament selection, where seven individuals are randomly selected from the population and the best is passed to the operator.

Fitness proportional selection was used in the experiments of chapters 3 and 4. This is a selection method which ensures that the best individuals in the population are chosen more frequently. However, it is possible that one individual could dominate the population after a few generations if it has a very high fitness function relative to the rest of the population. Tournament selection lessens the impact of one highly fit individual on

TABLE 5.3: Genetic programming initialisation parameters for the experiments of chapter 5

Population Size	1024
Generations	50
Crossover Proportion	90%
Reproduction Proportion	10%
Tree Initialisation Method	Ramped half-and-half
Selection Method	Tournament selection, size 7

the next generation, because the seven individuals from the population are chosen for each tournament regardless of their fitness.

The initialisation method for the individuals in the first generation has been changed to 'ramped half-and-half', which is the standard method used in the literature. This is in contrast to the 'grow' method, which was implemented for the experiments in the previous two chapters. Ramped half-and-half is a popular algorithm to generate trees, introduced in [164]. To generate a tree, it first chooses between the 'grow' and 'full' (the depth of every terminal node is equal) methods with equal probability. Then a tree is created with the chosen method, with a depth randomly chosen from between two and six nodes inclusive.

The Data Set

The data set generated for this chapter is similar to the data set used in chapter 3, which was defined by Falkenauer [96]. 20 training instances are randomly generated containing 500 pieces each, and the piece sizes are uniformly distributed between 20-100 inclusive. Every heuristic evolved for the experiments in this chapter is evolved on all 20 instances.

In total, 90 heuristics are evolved. 30 heuristics are evolved for their performance on the 20 instances. Another 30 heuristics are evolved on the first 250 pieces of each of the 20 instances. The final 30 heuristics are evolved on the first 100 pieces of each instance. Therefore, each heuristic is evolved over 20 instances, but some are only tasked with packing a subset of pieces.

The aim of this chapter is to test the evolved heuristics on problem instances much larger than those they are evolved on, to investigate how the heuristics' performance scales with the problem size. Thus, another set of 20 instances are generated, each with 100,000 pieces taken from the same 20-100 uniform distribution. The evolved heuristics are applied

to these instances, and the results are shown in section 5.3.

The performance of the evolved heuristics is presented in the results section in terms of the number of bins that the heuristic needs to pack all of the pieces. This is recorded at checkpoints, the intervals between which increase in size as more pieces are packed, giving more detail at the beginning of the packing and less at the end. For example, the number of bins used by the heuristic is recorded when each of the first ten pieces is packed, then at every ten pieces between ten and 100. At the end of the packing the checkpoints are every 10,000 pieces.

The Fitness Function

As tournament selection is the method for selecting individuals for the genetic operators, the fitness function must differentiate between two individuals' performance. A heuristic is deemed fitter than another if it requires less bins in total to pack all of the 20 instances. This provides a quick and reliable method of judging the relative fitness of two individuals.

If two heuristics that use the same number of bins are compared, then the fitness function from chapter 4 is applied. For ease of reference, it is shown again in equation 5.1, where n_j = number of bins used in instance j, $fullness_{ij}$ = sum of all the pieces in bin i of instance j, and C = bin capacity. The function iterates through 20 instances because each heuristic is evolved on 20 training instances, and the fitnesses of the solutions are summed to obtain the heuristic's fitness. Smaller fitness values are better. A fitness of 1000 is assigned to a heuristic that produces an illegal solution, an arbitrarily high value compared to the range of fitness values that a legal heuristic can produce.

$$Fitness = \sum_{j=1}^{20} \left(1 - \left(\frac{\sum_{i=1}^{n_j} (fullness_{ij}/C)^2}{n_j}\right)\right) \tag{5.1}$$

5.3 Results

30 heuristics are evolved on the first 100 pieces of the training set instances, 30 are evolved on the first 250 pieces, and a further 30 are evolved on all 500 pieces of each training instance. In this section, the three sets of 30 heuristics will be referred to as H_{100}, H_{250}, and H_{500}. After the heuristics have been evolved, they are applied to a set of 20 validation instances, each with 100,000 pieces. The validation set instances are unseen by the heuristics during their evolution.

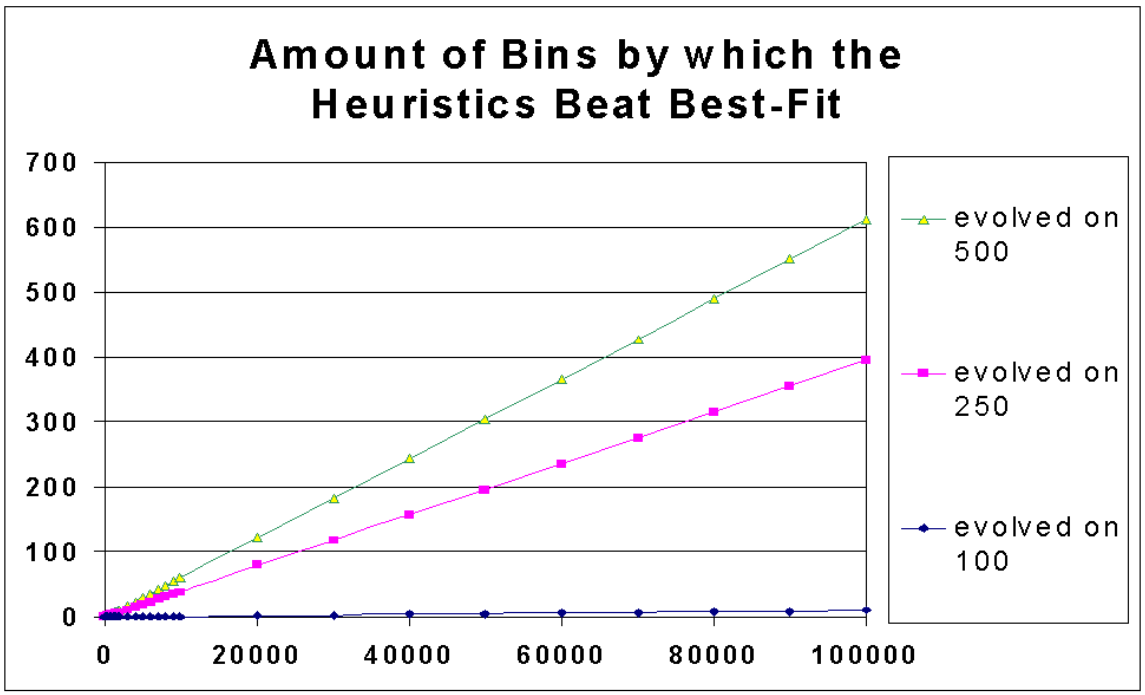

FIGURE 5.1: The results for H_{100}, H_{250}, and H_{500} on the validation instances. The vertical axis shows the amount of bins by which the 30 heuristics from each group beat best-fit on average. The horizontal axis shows the number of pieces packed. The results shown are the average amount of bins required by the heuristics over the 20 validation instances.

5.3.1 Performance past the training range

This section discusses the results of the heuristics over the full 100,000 pieces of each instance in the validation set. The results are displayed graphically in figure 5.1, which shows three series of results, one for each of H_{100}, H_{250}, and H_{500}. Recall that 30 heuristics are evolved for each of these classes. For each heuristic, we measure the number of bins used by the heuristic at each checkpoint for each of the 20 instances, and then record the mean average of the 20 values taken at each checkpoint (there will be one value taken at each checkpoint for each instance). There are 30 of these mean averages at each checkpoint (because there are 30 heuristics), so we calculate the average of these 30. This is then subtracted from the average number of bins used by best-fit at this checkpoint over the 20 instances, and the result is plotted in the graph of figure 5.1. In addition to this, table 5.4 shows the number of bins required on average by the 30 heuristics on each individual instance of the validation set (where each problem has 100,000 pieces).

TABLE 5.4: The number of bins used on average by the three groups of 30 evolved heuristics over the set of problems with 100,000 pieces, compared to the number used by best-fit

Instance	Best-Fit	H_{100}	H_{250}	H_{500}
1	41694	41677.70	41292.43	41070.00
2	41644	41637.83	41255.80	41032.57
3	41769	41757.30	41368.77	41148.37
4	41744	41735.83	41348.53	41135.83
5	41714	41696.43	41308.63	41088.33
6	41714	41703.87	41313.43	41094.03
7	41751	41732.77	41354.20	41128.57
8	41729	41722.20	41336.83	41116.50
9	41751	41735.97	41354.83	41136.03
10	41664	41648.67	41271.50	41058.33
11	41807	41797.70	41413.83	41201.53
12	41786	41770.20	41394.10	41179.57
13	41796	41781.97	41389.10	41168.80
14	41755	41759.43	41375.17	41165.00
15	41703	41697.33	41310.10	41089.27
16	41768	41766.00	41383.53	41166.67
17	41706	41707.03	41325.43	41107.07
18	41788	41776.20	41386.40	41169.63
19	41828	41817.40	41430.17	41217.73
20	41814	41808.40	41418.50	41197.97
Average difference from best-fit =		9.74	394.69	612.66

The results show that when applied to problems much larger than their training set, the evolved heuristics still maintain their performance, which is better than the performance of best-fit. The heuristics evolved on the first 100 pieces of all the training set instances use an average of 9.7 bins less than best-fit after 100,000 pieces have been packed. This may not seem like very many. However, a paired difference t-test over the 20 instances shows that the set of 20 average results, from the heuristics evolved on 100 pieces, is significantly better than the 20 results from best-fit. The results from the heuristic sets evolved on 250 and 500 pieces are significantly better to an even greater degree.

The heuristics evolved on 250 pieces perform better than the heuristics evolved on 100 pieces, requiring an average of 394.69 less bins on average. The heuristics evolved on 500 pieces perform better still, requiring 612.66 less bins. These results make it possible to conclude that heuristics perform better when they have been evolved on larger training instances. This highlights a trade off between the time taken to evolve the heuristics and their performance on new problems.

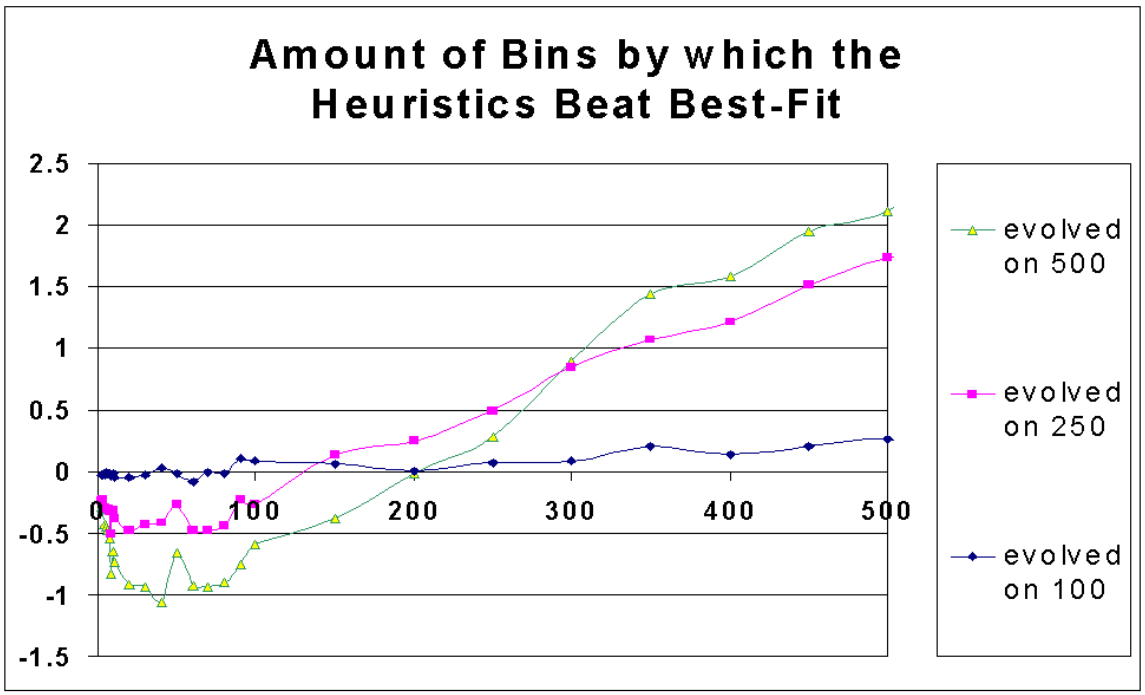

FIGURE 5.2: The results for the heuristics evolved on 100, 250, and 500 pieces of the training instances, on the first 500 pieces of each of the 20 validation instances. This is a 'zoomed in' view of the same data displayed in figure 5.1. The vertical axis shows the amount of bins by which the 30 heuristics from each group beat best-fit on average. The horizontal axis shows the number of pieces packed. The results shown are the average results of the 30 heuristics over the 20 instances.

5.3.2 Performance over the training range

This section will discuss the performance of the three sets of evolved heuristics on just the first 500 pieces of each validation instance. This provides some insight into *how* the heuristics are able to beat best-fit over the long term. Figure 5.2 shows a 'zoomed in' view of the graph in figure 5.1, and as such it displays the detail of the beginning of the packing.

H_{500} clearly opens more bins than is necessary when packing the first 200 pieces. This can be seen by the fact that the line goes immediately below the x-axis at the start of the packing, showing that best-fit uses less bins than the evolved heuristics do. However, when 250 pieces have been packed, the heuristics are using less bins than best-fit on average. This can be seen in figure 5.2 because the line crosses over the x-axis, and is above it when 250 pieces have been packed. Note that 250 pieces is halfway through the amount of pieces in the training set.

H_{250} also begins the packing by using more bins than best-fit. The graph line goes immediately below the x-axis and then crosses it when around 125 pieces have been packed. When a reading is taken at the 150 piece checkpoint, H_{250} is using less bins than best-fit on average. 125 pieces is half the amount of pieces in the training set for H_{250}. It seems that when packing the pieces at the beginning, these two sets of heuristics sacrifice short term efficiency to have more choice later on. This effect is less obvious in figure 5.2 for H_{100}, but the effect is still there. The line goes below the x-axis at the beginning, and is above the x-axis by the time 100 pieces have been packed. After this point, the heuristic is always better than best-fit.

To explain this behaviour it is necessary to refer to the concept of 'open' and 'closed' bins. A bin is said to be open if it has enough free space to be able to accommodate a further piece. Conversely, a bin is said to be closed if it does not. We can see whether a bin is closed because we know that the minimum size of a piece is 20. Therefore, for these problem instances, if a bin has less than 20 units of free space then it is said to be closed. Intuitively, a human would know that if many more pieces are still to be packed, it would be shortsighted to fill a bin to 131 fullness (19 free units of space). This is because the bin would then be closed, and 19 units of free space would be wasted. It would be more prudent to put the piece into another bin and wait for a piece that fills the bin to capacity. Such a piece may never arrive, but it is more likely to arrive if it is known that many more pieces are still to come.

The evolved heuristics do appear to have knowledge of the amount of pieces that they are asked to pack in their training sets, because they are beating best-fit by the time half of the number of pieces in their training set have been packed. Then they further increase their lead over best-fit at every checkpoint, and finish the packing having used less bins than best-fit. Of course, the heuristic cannot know the how far it is through the packing. It has no information given to it about how many pieces have been packed and therefore cannot know that it must pack more efficiently when it is halfway through the packing. The heuristic merely takes one piece and decides where to pack it, and it will use the same logic to make its decision at the start of the packing as it will at the end. Therefore, in contrast to how the results look, the heuristics do not wait until the end then begin packing efficiently, because they cannot know how close they are to the end.

To take H_{250} as an example, the results suggest that the heuristics keep many bins open at the start of the packing, to maintain more choice for where to put new pieces.

These bins are then closed only when a piece is found which fits very well into the remaining free space. When 250 pieces have been packed, there will *still* be many open bins, just as there were at the start. But less bins in total will be used than best-fit, because the bins that have been closed are filled more efficiently. This efficiency more than makes up for the inefficiency of the open bins that exist at that point. This is true to a greater extent for H_{500}, where the heuristics seem to have evolved to keep *more* bins open than the heuristics of H_{250}. It is also true to a lesser extent for H_{100}.

In summary, it seems that the heuristics of H_{500} are much more 'liberal' when deciding upon whether to open a new bin for a piece. They are less 'risk averse' in this sense, because during their evolution it was more likely for a piece to appear later that fits exactly into the gap in the open bin that was left free. This behaviour pays off when applied to much larger instances, because it is highly likely for a well fitting piece to appear before the packing is over. In contrast, the results suggest that the heuristics of H_{100} are more risk averse, and often choose to close bins 'prematurely'. This is because heuristics with the behaviour of H_{500} would receive low fitnesses, and be eliminated from the population, as the number of efficiently packed closed bins would not be able to compensate for the open bins when a small number of pieces have been packed.

Best-fit is a very good short term strategy, because it will never open a new bin unless it has to. Therefore, it would be very difficult to evolve a heuristic to beat it over an instance set containing many different sized instances. In such a scenario, the heuristic must pack as efficiently as possible all of the time because any piece could be the last. It would be inefficient to open a new bin just before the packing is complete, when an existing bin could have accommodated the piece.

5.3.3 Two Evolved Heuristics

This section will present two example evolved heuristics in graphical form. Both heuristics outperform best-fit on average over the 20 validation set instances. One is an example heuristic with a counterintuitive packing policy, and an interesting ridge structure. The second is a more complex heuristic, which appears to have arisen in response to the fact that there are no pieces smaller than size 20. The heuristics are compared to a plot of best-fit to highlight the differences.

Best-Fit

Within the framework that the heuristics are applied with in this chapter, the heuristic $C/(E-S)$ expresses best-fit. The C in this function can be any constant value, and E and S are the emptiness of the bin and the size of the piece respectively. This is because when a piece fits exactly into a bin, the heuristic will return the maximum value. $E-S$ will be zero, and the protected divide function will change this to 0.5, so the result will be $2C$.

The function is expressed graphically in figure 5.3. There is a 'wall' through the middle of the graph, behind which the results are negative, and in front of which the results are positive. The wall exists on the boundary where the piece size is equal to the emptiness, where the piece fits exactly into the bin. Behind the wall, the piece size is greater than the emptiness, so the piece does not fit into the bin. The result where the piece is put into an empty bin is always positive, and therefore the empty bin will always receive a higher score than any that would be overfilled. The heuristic will always prefer better filled bins because the values get higher the closer the piece size is to the emptiness.

Example Heuristic One

Our best evolved heuristic is displayed graphically in the same way in figure 5.4. The plot is rotated so that the large emptiness side is in the foreground, because it is this area of the graph where the heuristic differs from best-fit. This graph has a wall where the values are negative behind and positive in front. This, as in best-fit, is to prevent a bin from being overfilled, because the empty bin will always receive a higher score. In general, as the free space in the bin gets greater, the value returned for a bin reduces (for a given piece size).

However, this rule changes when the piece size is small and the emptiness is high. Figure 5.4 shows that the surface of the graph 'curls up' at the end, meaning that a high value is returned for combinations of a small piece size and large emptiness. This means that the heuristic will sometimes choose to put small pieces into very empty bins. In contrast, in the graph of best-fit this area of the plot gradually gets lower, meaning that the best fitting bin will always be chosen. The good performance of this evolved heuristic suggests that it is an advantage to maintain a larger diversity of fullnesses in the open bins. This affords the heuristic more choice when later pieces are packed.

Another difference between figures 5.3 and 5.4 is the raised 'ridge' in the function landscape of our best evolved heuristic (figure 5.4), running in front of and parallel to the

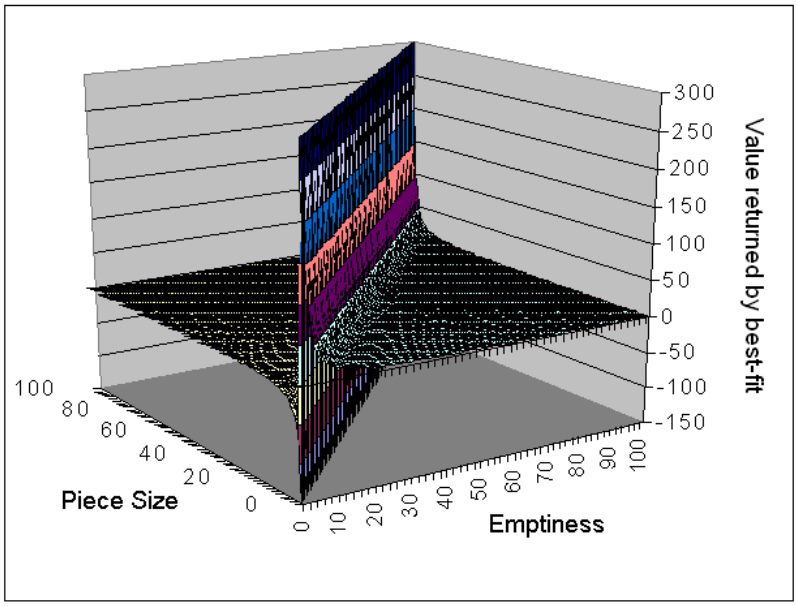

FIGURE 5.3: The function $C/(E-S)$, which represents the best-fit heuristic when applied in the fixed framework used in this chapter. The defining feature is the 'wall' which exists on the boundary of where the piece size matches the 'emptiness' of the bin. Behind this wall the piece size is larger than the emptiness, so the piece does not fit in the bin. The bin which receives the maximum score will be the bin with the least emptiness out of all of the bins that the piece fits into.

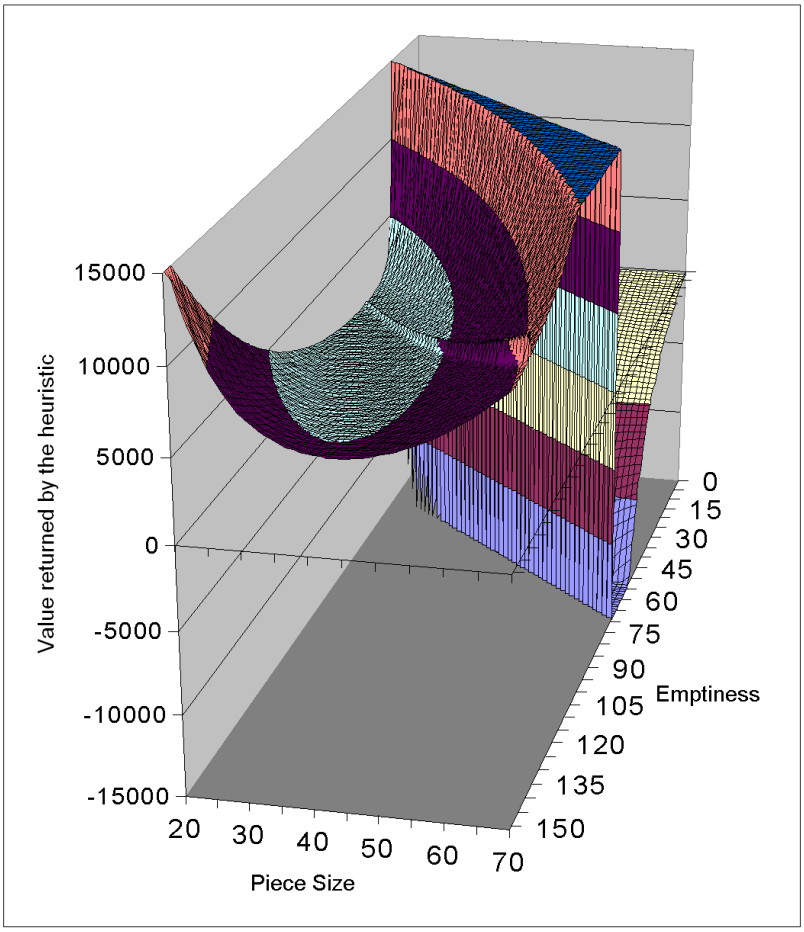

FIGURE 5.4: A heuristic requiring 41448.4 bins on average over the validation set, thus outperforming best-fit. The graph is in the same format as figure 5.3, but rotated to put the 'high emptiness' side in the foreground. The heuristic has much the same structure as best-fit, however the corner curls up at the end as the piece size gets small and the emptiness gets large. There is also a 'ridge' in the landscape running in front of the wall, where certain combinations of piece size and emptiness are rated higher than those adjacent to the ridge on both sides.

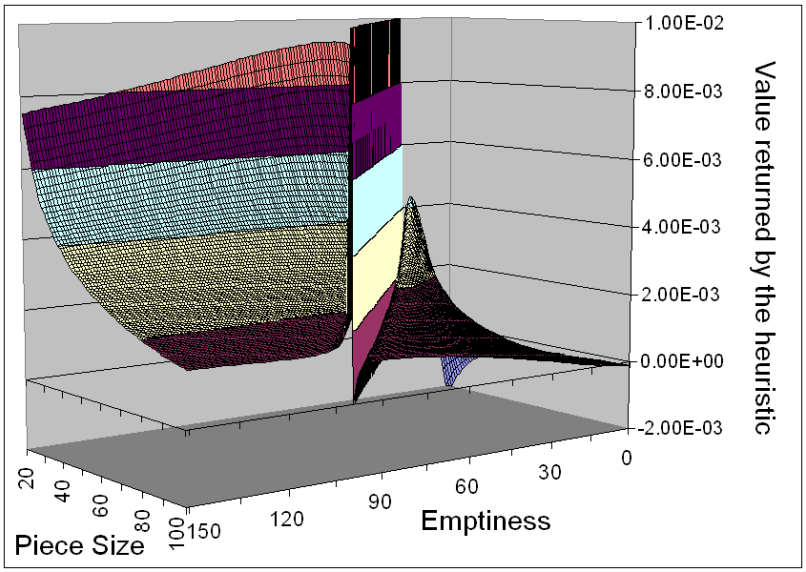

FIGURE 5.5: A heuristic requiring 40573.3 bins on average over the validation set, thus outperforming best-fit. This is the same heuristic as is displayed in figure 5.6

wall, from the left of the plot to the right. This means that there are certain combinations of piece size and bin emptiness that are rated higher than the combinations adjacent to them. The ridge occurs where the emptiness is twice the size of the piece. The landscape would smoothly curve downwards and then back up towards the wall, but for two values where the emptiness is twice the piece size and one more unit than this value. For example, if the piece size is 50, then the values returned by the heuristic when the emptiness is 100 and 101 do not fit the pattern. The value when the emptiness is 100 is more markedly different to the rest of the sequence than than when the emptiness is 101.

Example Heuristic Two

The second example heuristic is displayed in figure 5.5 and 5.6. These figures show the same graph but they are rotated 180° relative to each other, to display the other side of the plot. This heuristic has a wall in its fitness landscape, similar to best-fit, which prevents a bin being overfilled.

The side of the wall where a piece would be too large for a bin is on the right of the wall in figure 5.5, and on the left of the wall in figure 5.6. The function forms an interesting landscape on this side of the wall, but the shape is irrelevant because there will always be an empty bin available to the heuristic. For a given piece size, the value returned for

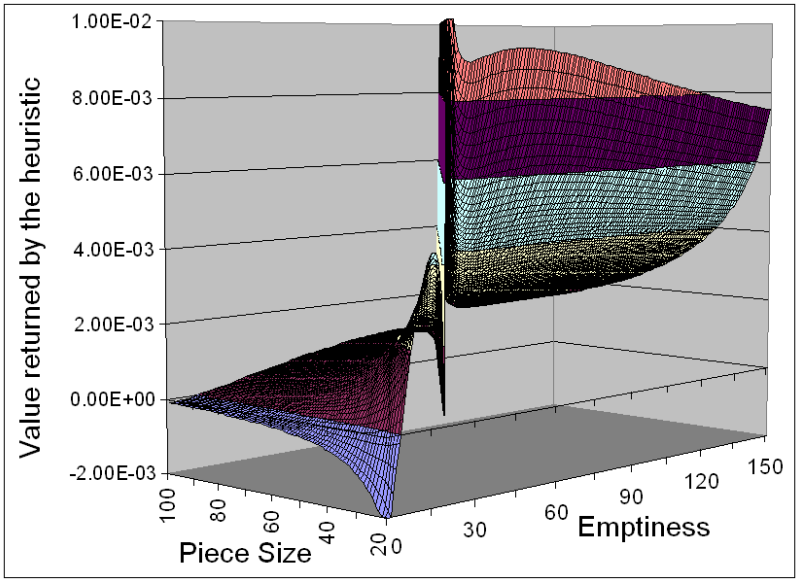

FIGURE 5.6: A heuristic requiring 40573.3 bins on average over the validation set. This is the same heuristic as is displayed in figure 5.5

the empty bin (emptiness 150) is always higher than the maximum value of any emptiness behind the wall.

The function of the heuristic, therefore, must be attributed to the shape of the landscape in front of the wall. This is to the left in figure 5.5, and to the right in 5.6. This area is where the piece size is less than or equal to the emptiness. If the piece fits exactly into the bin then the value returned is high (off the scale of the top of the graph), so if such a bin is available the piece will be put there.

The interesting feature of this heuristic is the section of the graph where the emptiness is between the piece size (S) and roughly double the piece size (roughly $2S + 5$). Between these values there is a dip in the graph, where the values are lower than those either side. The dip reaches a low point at roughly 20 units of emptiness greater than the size of the piece.

For example, a piece of size 20 will be put into a bin with an emptiness of between 20-24 over all others. As the emptiness increases, the values returned by the heuristic gradually get lower until a low point is reached at an emptiness of 28. Then the values rise again until 48 emptiness, and then start a decline that continues until the full 150 capacity of the bin. This means that if there is a group of bins that the piece will nearly fill, the

heuristic definitely wants to avoid leaving a gap of around eight after the piece is placed, and if it can avoid leaving any gap of between 5-27 then it will.

To give another example, a piece of size 70 will be put into a bin with 70 emptiness if one is available. The heuristic would choose a bin between 70-75 emptiness if there are any available, because such bins receive the highest scores for a piece of size 70. For emptinesses from 71-90 the values returned get lower, down to a low point at 90. After this the values get higher again slowly until reaching a high point at 145, and then the values get lower again for the remaining five units of the 150 capacity.

To summarise these two examples, they show that the heuristic chooses a bin where the piece fits exactly. If none are available, it then chooses a bin where four or five units of space would be left at the top. Then if none of these are available it will choose a bin where the piece would fill roughly half the emptiness of the bin (leaving a lot of space). For any given piece size, some of the least favoured bins are those which leave a gap that no further piece can fill, because none are smaller than 20. This will have the effect of maintaining a larger variety of bins that are open, and the extra choice that the heuristic has means that it can be more efficient in the long run.

5.4 Conclusion

The aim of this chapter was to show that heuristics can be evolved on small problem instances and maintain their performance on problem instances orders of magnitude larger than those they were trained on. This has been shown by creating a training set with 500 pieces in each instance. 30 heuristics were evolved by their performance packing the full 500 pieces of each instance, each in a separate genetic programming run. In addition to this, 30 were evolved only packing the first 250 pieces of the instances, and 30 were evolved on the first 100 pieces. Each of the 90 evolved heuristics were then applied to a set of instances each with 100,000 pieces and these results have been presented in this chapter.

Compared to best-fit, the evolved heuristics have a tendency to keep more bins open. This allows better performance because, over many pieces, these open bins can be filled more efficiently. This strategy has the effect of under performing when the first set of pieces is packed, but by the end of the packing less bins are used in total. The effect of under performing at the start of the packing is an example of a long term strategy, which has arisen even though the heuristics themselves have no knowledge of how much of the

packing has been completed as they pack the current piece. It has also been shown that the heuristics are more likely to allow more bins to be opened at the start of the packing if they have been evolved with more pieces in the training set instances. The heuristics generally overtake the performance of best-fit when half of this number of pieces have been packed. It is analogous to saying that the heuristics are less 'risk averse' when they have been evolved packing more pieces. When fewer pieces are used in the training set, the heuristics are much more likely to have behaviour which is effective in the short term, as if they are unwilling to risk opening new bins in case the next piece is the last.

The heuristics evolved for one dimensional bin packing are very quick, taking seconds to solve a given instance of reasonable size. The heuristics, as applied here, take in the order of hours to fully pack a 100,000 piece instance. This is a large value because of the heuristic being applied to *every* bin in the solution, even the completely full bins and those which could not possibly fit a piece inside because no piece is smaller than 20. This time could easily be reduced by several orders of magnitude, by not evaluating the bins which evidently cannot receive a piece. For example, assuming roughly three pieces fit into a bin on average, to pack the last piece of a 100,000 piece instance requires over 33,000 evaluations of the heuristic. A small fraction of those would be open bins which would actually be able to accommodate the piece, and these are the only bins that it is productive to consider.

The time issue has an important implication. Assuming that a heuristic could pack an instance very quickly after it has been evolved, the results that have been presented in this chapter show that it is possible to evolve a heuristic on a reduced problem, and then apply it to solve a larger problem, providing that the reduced problem is still representative. The bulk of the time taken with this hyper-heuristic method is in the evolution stage. An evolutionary algorithm operating on a space of *solutions* may take a long time to obtain a solution for twenty 100,000 piece instances. We conclude that the hyper-heuristic paradigm, while automating the heuristic design process, also has the potential to alleviate, to some extent, the problem of combinatorial explosion, where the time taken to obtain a solution increases exponentially with the size of the problem.

To summarise chapters 3, 4, and 5, heuristics for one dimensional bin packing have been automatically designed and built by our genetic programming hyper-heuristic methodology. We have shown that they are re-usable in certain circumstances, and that they maintain their performance on new problems much larger than their training set. In

the next chapter, the problem domain is changed to the two dimensional strip packing problem. The functions and terminals will be changed from those familiar in the previous three chapters, and the framework within which the heuristics are applied is also changed, because of the representation of the problem in two dimensions. However, the methodology will remain essentially unchanged, applying a heuristic function which scores the potential locations for each piece. The similarities and differences are discussed more thoroughly in the introduction to the next chapter.

Chapter 6

Evolving Heuristics for Two Dimensional Strip Packing

6.1 Introduction

The previous three chapters have presented methodologies to evolve heuristics for one dimensional bin packing. This chapter presents a framework and representation to enable automatic heuristic generation for the two dimensional strip packing problem. While the representation is necessarily different, the fundamental packing methodology is the same as for chapters 3, 4, and 5. A heuristic is evaluated once for each potential allocation, and the allocation that receives the highest score is performed.

The two dimensional strip packing problem is explained in the introduction to this thesis, in section 2.4.1, but a brief description is given here for ease of reference. A set of two dimensional shapes must be packed into a two dimensional container, which has both a width and an infinite height. This is analogous to cutting a required set of shapes from a sheet of material. The objective is to minimise the height of the sheet that is needed to accommodate all of the shapes, and thus minimise the waste material when a guillotine cut is made across the width of the sheet. The problem arises in real world industrial problems when rolls of material are used in manufacturing. For example, clothing manufacture requires shapes to be cut from stock rolls of textiles, and window panes must be cut from stock sheets of glass.

There are three differences between the methodology presented in this chapter and that of the previous three chapters. The first difference is that the heuristics have a choice of which piece to pack, from all of those available. In the previous methodology, a heuristic

packed each piece in turn, in the order in which they appeared in the piece list. In this chapter, the heuristics can choose any piece from the list of those remaining, *and* choose where to pack it. This changes the problem from online to offline, because the heuristic has knowledge of all of the pieces before the packing begins.

The second difference concerns the ability of heuristics to produce illegal solutions. In chapters 3, 4, and 5, a bin could be overfilled by a heuristic, resulting in an illegal solution. This was permitted so that the heuristics could learn the rules of the problem, as well as evolving a good packing strategy. For the two dimensional problem representation there is no similar way to allow the heuristics to violate the hard constraints of the problem. In addition to this, the representation of the problem domain is complex enough that trivial heuristics do not produce good solutions. This is in contrast to the representation of the one dimensional bin packing problem, where a heuristic consisting of just one terminal could produce the same result as the first-fit heuristic if illegal solutions were permitted. For these reasons, illegal solutions are prevented in the experiments of this chapter. That is, the heuristics can only consider legal allocations of pieces.

In the previous three chapters, the heuristics evolved for one dimensional bin packing were shown to be re-usable under certain circumstances. In this chapter, the heuristics are treated as disposable, in the sense that they are evolved to produce a solution to one problem instance. The heuristics *can* be reused on unseen instances, but they may not produce the same quality of results as they do on the instance they are evolved for. The hyper-heuristic can therefore design a heuristic for an individual instance, allowing the heuristic to perform better than any one comparable constructive heuristic that performs generally well across a wide range of instances.

In the results section, we show that this hyper-heuristic methodology produces results better than human created metaheuristics, and the best available constructive heuristic. The results are also compared to the state of the art reactive GRASP methodology [4]. However, the complexity of the GRASP heuristic is far beyond what a computer could currently design. The results of this chapter show that it is possible to automate the heuristic design process for two dimensional strip packing, and that the quality of the evolved heuristics is comparable to that of heuristics designed by humans.

The chapter is organised as follows. Section 6.2 explains how the sheet is represented, and where the pieces can be placed in the solution. Section 6.3 explains how the heuristic selects a piece and where to place it, and how the heuristics are evolved. Section

6.4 displays the results, and section 6.5 presents the conclusions to the chapter.

6.2 Representation

Constructive two dimensional strip packing heuristics treat the problem as a sequence of steps, placing a piece into the solution at each step. The heuristics evolved in this chapter decide both which piece to place and where to place it. The potential locations for a piece are described in this section.

The available spaces on the sheet are determined by a number of 'bins', the number and configuration of which is defined by the arrangement of the pieces that are already in the bin. Each bin has a width, a lateral position, and a height in relation to the base of the sheet. This representation takes its inspiration from the paper by Burke, Kendall and Whitwell [38], which introduces the best-fit heuristic for two dimensional packing. This heuristic selects the lowest available space, and then finds the piece whose width best fits it. In order to efficiently find the lowest space, the sheet could be represented as a dynamic set of bins.

At the beginning of the packing there is one bin, which has a width equal to the width of the sheet, and a height equal to zero, because its base is the base of the sheet. The heuristic can choose to place a piece in either the left or right corner of a bin, so at this initial stage there are two locations into which a piece can be placed. In the majority of cases, when a piece is put into the solution, it will split the bin into two separate bins. One will appear next to the piece, representing the remaining space in the initial bin. The second will form on top of the piece, and will have a width equal to the width of the bin. This is the most common modification of the bin structure. However, bins can merge if they are adjacent and their height is equal, and bins can be filled if they are too narrow for a piece.

As an example, figures 6.1-6.4 show a step by step example of packing two pieces. Figure 6.1 shows the initial partial solution with two pieces packed into the lower left and right corners. The bin structure is shown in each diagram to illustrate how it is modified after each piece is placed. The four situations where the bin structure will change are described in more detail in the following four sub-sections.

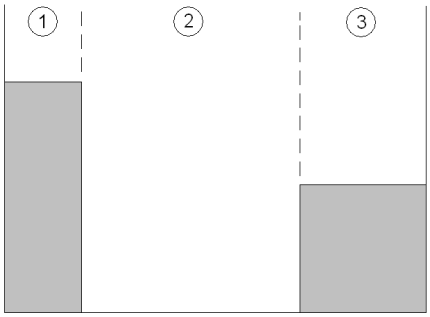

FIGURE 6.1: Bin structure after two pieces have been packed

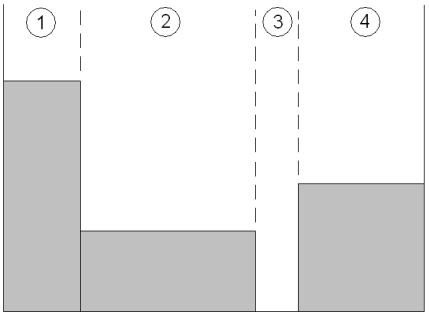

FIGURE 6.2: The middle bin of figure 1 has been replaced by two new bins due to the third piece being packed into it

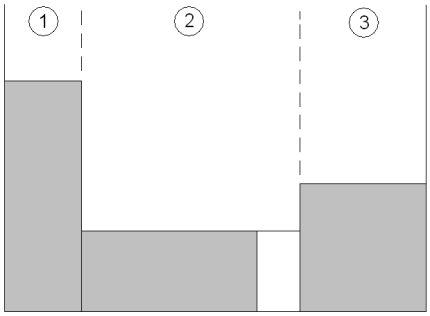

FIGURE 6.3: As no piece can fit into bin three of figure 2, the base of bin three has been raised, to combine it with bin two

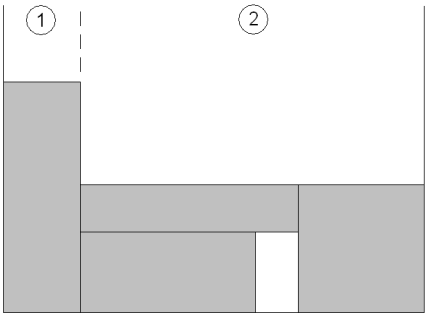

FIGURE 6.4: When the fourth piece is packed into the second bin, the heights of bin two and three are the same and they have been combined

Bin creation

When a piece P is placed in bin B_x, B_x is replaced by two bins, B_y and B_z. B_y will form directly on top of P, and B_z will represent what remains of B_x, and will be formed only if $Width(P) < Width(B_x)$. Formally:

$$Height(B_y) = Height(B_x) + Height(P)$$
$$Width(B_y) = Width(P)$$
$$Height(B_z) = Height(B_x)$$
$$Width(B_z) = Width(B_x) - Width(P)$$

This process is shown in figures 6.1-6.2.

Bin Combination

Using the notation from the bin creation section, if the height of B_y is equal to the height of the neighbouring bin B_n, forming a continuous flat level, then B_y is joined to B_n, amalgamating them into one bin. This process is shown in figure 6.4.

Raising Narrowest Bin

If B_z is too narrow for any remaining piece to fit in, then $Height(B_z)$ is increased to the level of its lowest neighbour, and then the bin combination process occurs between the two bins. This is *only* invoked if B_z is between two higher bins. The space is wasted space, and so can be discarded for the purposes of continuing to construct a solution. If B_z is not between two higher bins, then the space cannot yet be counted as wasted space. This process is shown in figures 6.2-6.3.

Raising Lowest Bin

If there is no piece that can fit in any bin, the lowest bin in the current solution is raised to the level of its lowest neighbour, and the two bins are combined, forming a wider bin. This is done iteratively until a bin is formed that is wide enough to accommodate at least one piece.

6.3 Methodology

The previous section explained the representation of the two dimensional bin packing problem. Section 6.3.1 explains how this representation is manipulated by a heuristic in order to produce a solution. Section 6.3.2 then describes the structure of the heuristics. Using information given in these two sections, an example of the packing process is worked through in section 6.3.3. Finally, section 6.3.4 presents the genetic programming parameters, the data sets used in the results section, and the fitness function employed to assess the relative fitness of the heuristics.

6.3.1 How the Heuristic is Applied

This section describes the process by which the heuristic is applied to an instance to obtain a solution. As in chapters 3, 4, and 5, the heuristic operates within a fixed framework. The framework for this chapter is shown in algorithm 5. In the previous three chapters, the heuristics returned a value for each location available for the piece that it was packing at the time. The piece was placed at the location which received the highest value. Due to the change in representation, the framework is naturally different to that of previous chapters. The underlying method, of scoring each possible place for each piece, is unchanged.

However, there are two fundamental differences. Firstly, in previous frameworks, every location was scored, even those into which the piece could not fit. If the heuristic resulted in an illegal solution because of this, it was penalised heavily by the fitness function. In this chapter, illegal solutions are prevented from occurring. Every heuristic will produce a legal solution because the framework in which they operate only allows legal placement of pieces. This is achieved with a modification of the framework, where the heuristic is now only evaluated where the allocation of the piece and location would be a legal one.

The second difference is that the heuristic decides which piece to pack, as well as where to place it. This is in contrast to the framework in the previous chapters, where the pieces are packed one at a time, in the order that they appear in the piece list. The heuristic only had to decide *where* to place the given piece. This is a fundamental difference, because the problem becomes offline rather than online. All of the pieces are known before the packing starts, and they can be packed in any order.

The pseudocode for the framework that applies the heuristics is shown in algorithm 5. It refers to the concept of an 'allocation'. In each bin (recall that a bin is part of the

Algorithm 5 Pseudo code of the framework within which the heuristics are applied in the experiments of chapter 6

```
1: while pieces exist to be packed do
2:   if narrowest bin is too narrow for any piece AND narrowest bin is between two higher
     bins (or the sheet edge) then
3:     raise narrowest bin to level of lowest neighbour
4:   else if at least one piece can fit in any bin then
5:     for each bin b in partial solution do
6:       for each piece p in pieceList do
7:         for each orientation o of piece p do
8:           if piece p fits into bin b then
9:             current_allocation = b,p,o
10:            heuristic_output = evaluateHeuristic(current_allocation)
11:            if heuristic_output > bestSoFar then
12:              bestSoFar = heuristic_output
13:              best_allocation = current_allocation
14:            end if
15:          end if
16:        end for
17:      end for
18:    end for
19:    perform the best_allocation on the partial solution
20:  else
21:    raise lowest bin
22:  end if
23:  update bin structure
24: end while
```

representation of the sheet), a piece can be put in the bottom-left corner or in the bottom-right corner. There are also two orthogonal orientations that the piece can adopt in both corners. Given a partial solution, we will refer to a combination of piece, bin, corner and orientation as an allocation. Therefore, there is a total of four allocations to consider for each piece and bin combination, provided that the piece's width in each orientation is smaller than the width of the bin.

Line 1 of the algorithm shows that the framework iterates until all of the pieces have been packed. The conditional statement, on line 2, checks if the narrowest bin in the sheet is too narrow to accommodate any remaining piece. If it is also between two higher bins then it is 'filled' up to the height of the lowest of the two. This process is described as 'raising the narrowest bin' in section 6.2, and shown in figure 6.3.

If the narrowest bin can accommodate a piece then we can be sure that at least one piece can fit into a bin, and so the main body of the code executes, from line 5 to line 19. The heuristic is evaluated once for each allocation that does not result in an illegal solution. A value will be returned by the heuristic for each allocation, which is treated as its 'score'. This is shown on line 10 of the algorithm. A higher score means that the heuristic rates an allocation as better. The allocation which receives the highest score is the one that is performed on the partial solution on line 19, because the heuristic has chosen it from all of the options available. In other words, the piece from the allocation is put into the bin from the allocation, in the orientation dictated by the allocation.

The algorithm shows, on line 21, that if no piece can fit into a bin, then the lowest bin is raised. This is explained in section 6.2 under the heading 'raising the lowest bin', and by figure 6.4. On line 23 the bin structure is updated, making sure that adjacent bins of the same height are amalgamated into one, and initialising new bins where a piece has been placed into the solution.

6.3.2 The Structure of the Heuristics

The functions and terminals used for the experiments in this chapter are shown in table 6.1, with descriptions of each. The terminals are shown in the lower six rows of the table. The heuristic is evaluated for each allocation, which represents a valid potential piece, orientation, and bin combination (see section 6.3.1). There are two terminals representing the size of the piece, two representing the bin of the current allocation, and two representing

TABLE 6.1: The functions and terminals used in chapter 6, and descriptions of the values they return

	Name	Label	Description
Functions	Add	+	Add two inputs
	Subtract	-	Subtract second input from first input
	Multiply	*	Multiply two inputs
	Divide	%	Protected divide function
Terminals	Width	W	The width of the piece
	Height	H	The height of the piece
	Bin Width Left	BWL	Difference between the bin and piece widths
	Bin Height	BH	Bin base's height, relative to base of sheet
	Opposite Bin	OB	Bin height minus height of opposite bin
	Neighbouring Bin	NB	Bin height minus height of neighbouring bin

the position of the piece in the bin.

The W terminal returns the width of the piece, and the H terminal returns its height. The values that these return will change depending on the orientation of the piece. The width is always the horizontal component, and the height is the vertical component. These two terminals allow the heuristic to determine the relative size of the piece, and enable the heuristic to distinguish between the two orientations of a piece if all other elements of the allocation are the same.

The BWL terminal stands for the bin width left. It returns the width of the piece subtracted from the width of the bin, and as such it gives the heuristic a measure of how well the piece fits into the bin. The BH terminal returns the bin height, which is measured as the distance between the base of the bin and the base of the sheet.

The 'opposite bin' and 'neighbouring bin' terminals give information about the difference in height between the current bin and either of the two bins adjacent to it. An allocation is either in the lower-left or lower-right of the bin. If the allocation is in the lower-left, then the neighbouring bin is the bin to the left of the current bin, and the opposite bin is to the right. If the allocation is in the lower-right of the bin then these labels are reversed. These terminals give the context of the current bin within the solution. For example, they can provide information on whether the bin is between two higher bins, or between two lower bins. Importantly, they also enable the heuristic to distinguish between an allocation with the piece in the lower-right corner and one in the lower-left corner of a given bin.

The four functions are the standard arithmetic operators. They all have identical functionality to the previous chapter, with the exception of the protected divide function.

TABLE 6.2: The values returned by the standard protected divide function, the modified protected divide used in the previous chapter, and the further modified function used in this chapter.

Calculation	$\frac{10}{1000}$	$\frac{10}{100}$	$\frac{10}{10}$	$\frac{10}{1}$	$\frac{10}{0.1}$	$\frac{10}{0}$
When % returns 1	0.01	0.1	1	10	100	1
When denominator is changed to 0.5	0.01	0.1	1	10	100	20
When denominator is changed to 0.001	0.01	0.1	1	10	100	10000

The protected divide function exists to ensure that when a division by zero occurs, a value is returned rather than an error. The standard protected divide function, as defined by Koza [164], returns a value of one. Recall that, in chapter 5, the protected divide function was modified from its standard functionality, changing the denominator to 0.5 if it was zero, and therefore effectively doubling the numerator. It was modified in this way because the values returned by the function did not follow a pattern (see section 5.2.2, and table 5.2, for a full explanation). The middle row of table 6.2 shows that if the denominator sequence between one and zero is expanded, for example by including 0.1, the pattern is again broken. The values go from 1 to 10 to 100 and then back down to 20 at zero. Table 6.2 shows that if the denominator is changed to 0.001 instead of 0.5, then the pattern is restored. This is the functionality of the protected divide function used in the function set for this chapter.

There will always be a point where the pattern of rising values will break. For example, if the denominator is 0.0001, the value returned by the function will be 100000. However, the table shows that, when the denominator is zero, the value returned is 10000, so this is again a drop in the value at the end of the sequence.

A possible solution would be to change the functionality of the protected divide function, so that a division by zero returns $+\infty$. This value is greater than would be returned by the protected divide function if the denominator was changed from zero to an arbitrarily small value, and therefore maintains the pattern in all circumstances. However, to return $+\infty$ may result in subtlety being lost in calculations elsewhere in the heuristic's tree structure. For example, adding, subtracting, or multiplying any value by ∞ will return ∞, meaning that the heuristic will return ∞ regardless of calculations that other branches of the tree perform. The 0.001 value is a compromise between maintaining the pattern as the denominator reaches zero, and maintaining the importance of other calculations elsewhere in the tree when a division by zero occurs.

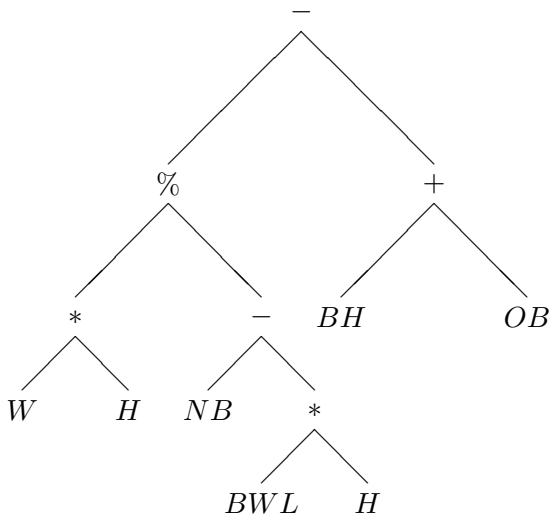

FIGURE 6.5: An example heuristic, which is analysed in section 6.3.3 by packing the pieces of figure 6.6

6.3.3 Example of the Packing Process

This section, which goes into more detail than section 6.3.1, works through an example of how the heuristic chooses a piece from those which remain to be packed, and then where to place it in the partial solution. The heuristic we will use in this example is shown in figure 6.5. It consists of nodes from the GP function and terminal set shown in table 6.1. It contains at least one of each of the functions and terminals, however this is not a requirement. A heuristic in the population could contain only a subset of the functions and terminals that have been made available.

We will use the heuristic shown in figure 6.5 to choose a piece from those which remain to be packed (shown in figure 6.6) and choose where to place it in the partial solution shown in figure 6.7. The partial solution shows that two pieces have already been packed by the heuristic, one to the left and one to the right. We do not show this process, because the example will be more descriptive if we start with a partial solution. There are three bins in the partial solution, labelled 1, 2 and 3, which are defined by the pieces already packed.

The framework takes each piece in sequence, and evaluates the heuristic for every possible allocation of that piece. So, first we will consider piece 1 from figure 6.6. A piece can be placed in the left or right of a bin, in either orientation, providing it does not exceed

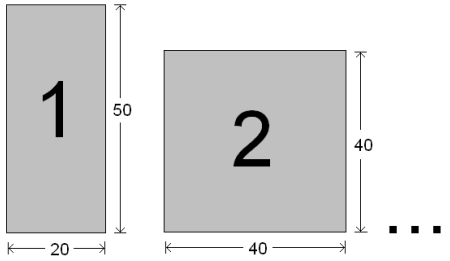

FIGURE 6.6: The pieces that the heuristic in figure 6.5 will pack for the example described in section 6.3.3

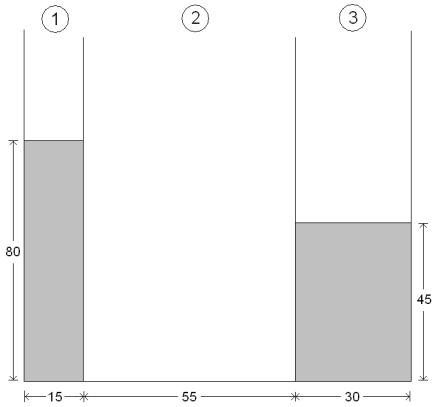

FIGURE 6.7: The current partial solution, the heuristic will choose a piece from figure 6.6 to be placed in this solution

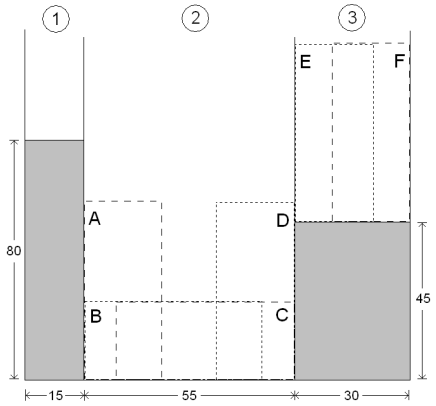

FIGURE 6.8: All of the locations where piece one from figure 6.6 can be placed

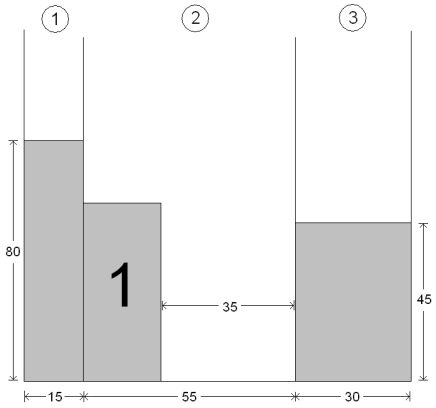

FIGURE 6.9: Allocation A from figure 6.8 in detail

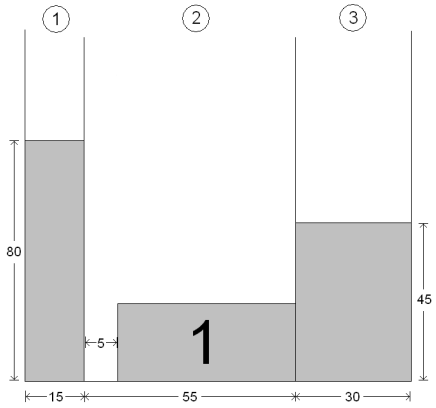

FIGURE 6.10: Allocation C from figure 6.8 in detail

the width of the bin. So, piece 1 cannot be placed into bin 1, because that bin is only 15 units wide, and the piece is 20 units wide in its narrowest orientation. Piece 1 can be placed into bin 3, but only in one orientation. The piece will be too wide if it is placed in its horizontal orientation. Bin 2 is wide enough to accommodate the piece in either of its orientations.

Figure 6.8 shows these six valid allocations for piece 1 in the partial solution, labelled A to F. Each of these six allocations will receive a score, obtained by evaluating the tree once for each allocation. The tree will give a different score for each allocation because the GP terminal nodes will evaluate to different values depending on the features of the allocation.

The process of evaluating the heuristic for allocations A and C is shown here. Figure 6.9 shows allocation A in detail. To evaluate the tree for allocation A, we will first determine the values of the terminal nodes of the tree. The 'width' (W) and the 'height' (H) terminals will take the values 20 and 50 respectively. The 'bin width left' (BWL) terminal will evaluate to 35 because that is the horizontal space left in the bin after the piece is placed. The 'bin height' (BH) terminal evaluates to zero, because the base of bin 2 is at the base of the sheet. The opposite bin for the allocation is bin 3, as it is to the right of bin 2, and the allocation is in the lower *left* corner of bin 2. The 'opposite bin' (OB) terminal will evaluate to -45, because the opposite bin has a height of 45, and OB is the height of the current bin minus the height of the opposite bin. The neighbouring bin is bin 1, because it is to the left of bin 2, and the allocation is also in the left of bin 2. The 'neighbouring bin' (NB) terminal will evaluate to -80, because the neighbouring bin has a height of 80.

The tree is displayed as an expression in 6.1. Expression 6.2 shows the same expression with the terminal values are substituted in. This simplifies to expression 6.3, which evaluates to -44.454 to three decimal places. This value is the score for the allocation of piece 1 in bin 2, in a vertical orientation and on the left side of the bin.

$$\left(\frac{W * H}{NB - (BWL * H)}\right) - (BH + OB) \tag{6.1}$$

$$\left(\frac{20 * 50}{-80 - (35 * 50)}\right) - (0 - 45) \tag{6.2}$$

$$-\left(\frac{1000}{1830}\right) + 45 \tag{6.3}$$

Figure 6.10 shows allocation C in detail. Again, we will calculate the values of the terminal nodes for this allocation in order to evaluate the heuristic. W and H are now 50

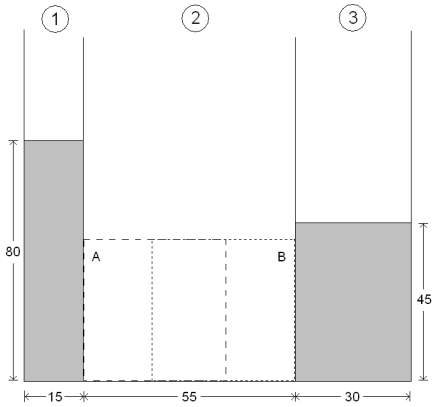

FIGURE 6.11: Both of the potential allocations for piece two

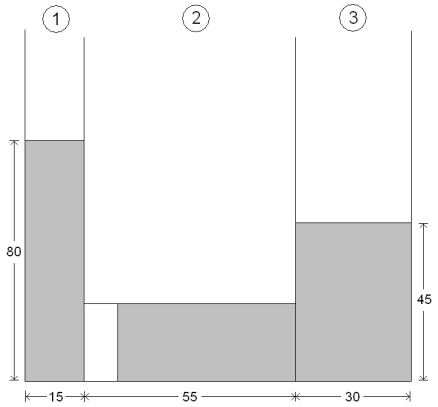

FIGURE 6.12: The new partial solution after the allocation which received the highest score has been performed

and 20 respectively. They are different from their values in allocation A because the piece is now in the horizontal orientation. BWL evaluates to five, as this is the horizontal space left in the bin after the piece has been placed, shown in figure 6.10. BH evaluates to zero, as before, because the allocation concerns the same bin. OB and NB evaluate to -80 and -45 respectively, as the allocation is for the lower right side of the bin rather than the left.

When the terminal values have been substituted in, the heuristic simplifies to expression 6.4, which evaluates to 73.103 (to 3 d.p.). This is the score for the allocation of piece 1 in bin 2, in a horizontal orientation and on the right side of the bin.

$$-\left(\frac{1000}{145}\right) + 80 \tag{6.4}$$

Of these two allocations we have shown, the second allocation has been rated as better by the heuristic because it received a higher score. The other four allocations for this piece are scored in the same way. Then the allocations possible for piece 2 are scored, of which there are essentially two, shown in figure 6.11. There are in fact four allocations which are scored for the second piece, but it has identical width and height so both orientations will produce the same result from the heuristic.

The rest of the pieces that remain to be packed have all of their allocations scored in the same way. Finally, the allocation which received the highest score from the heuristic is actually performed. In other words the piece from the allocation is committed to the partial solution, in the bin, orientation and position dictated by the allocation. Then the

TABLE 6.3: Initialisation parameters of the experiments of chapter 6

Population size	64, 128 and 256
Generations	50
Crossover probability	0.85
Mutation probability	0.1
Reproduction probability	0.05
Tree initialisation method	Ramped half-and-half
Selection method	Tournament selection, size 7

bin structure is updated according to the rules explained in section 6.2 because a new piece has been put into a bin.

For example, the allocation involving piece one in position 'C' from figure 6.8 received a score of 73.103. If no subsequent allocation (involving the same piece or any other piece) receives a higher score than this, then this will be the allocation that is performed. Assuming there are no further pieces to be packed that have a width smaller than five, the bins will then be reconfigured as shown in figure 6.12, because of the process of raising the narrowest bin, shown in section 6.2 and figures 6.2 and 6.3.

The process of choosing a piece and where to put it is now complete, and the next iteration begins. All the remaining pieces are scored again in the same way, and there will be new positions available due to the change in bin structure that has occurred.

6.3.4 How the Heuristics are Evolved

The process by which a heuristic packs a solution has been explained in the previous section. This section explains how the heuristics are evolved, and contains three sub-sections. The genetic programming parameters are explained in the first part. The data sets used in the experiments in this chapter are explained in the second part. The third part presents the fitness function employed to assign a fitness value to a heuristic.

Parameters

Table 6.3 shows the parameters used for the genetic programming algorithm. The experiments of this chapter are coded using the ECJ (Evolutionary Computation in Java) package, found at http://www.cs.gmu.edu/∼eclab/projects/ecj/. Three population sizes are tested, to investigate the extent to which the population size affects the quality of the evolved heuristics. The three values of 64, 128, and 256 are smaller than the standard 1000 popula-

tion size that was used in the one dimensional bin packing experiments. This choice reflects the need to keep the run times to reasonable levels.

A mutation operator is included in the set of genetic operators, which deletes a random subtree and then replaces it with a new subtree using the 'grow' method, with a minimum and maximum depth of five. The crossover operator produces two new individuals with a maximum depth of 17. Approximately 10% of the population of a new generation is produced using the mutation operator. 85% is produced using crossover, and 5% by the reproduction operator. As in the previous chapter, the trees are initialised with the ramped half-and-half method, and the selection of individuals is performed by tournament selection.

The Data Sets

The four benchmark data sets from the literature are explained in this section. The instances are summarised in table 6.4. All of the instances are created from known optimal packings, and are well studied in the literature, each having been used to benchmark the performance of a number of heuristic and metaheuristic methodologies. The use of these data sets means that we can compare the results of the hyper-heuristic methodology against a wide variety of other methodologies.

The Hopper and Turton data set contains seven classes of three problems each. Each class was constructed from a different sized initial rectangle and contains a different number of pieces. All pieces have a maximum aspect ratio of seven, meaning that the length of the longer side is never more than seven times the length of the smaller side. Valenzuela and Wang created two classes of problem, referred to as 'nice' and 'path'. The nice data set contains pieces of similar size, and the path data set contains pieces that have very different dimensions. The data set from Burke, Kendall and Whitwell contains twelve instances with increasing numbers of pieces in each. All of these datasets can be found at http://dip.sun.ac.za/∼vuuren/repositories/levelpaper/spp[1].htm. We also use an instance created by Ramesh Babu and Ramesh Babu, containing 50 rectangles all of similar size. The dimensions of the pieces in this instance are given in [233].

The Valenzuela and Wang data set uses floating points to represent the dimensions of the rectangles. The implementation of this hyper-heuristic methodology uses integers, so to obtain a data set we can use, the data is multiplied by 100 and rounded to the nearest integer, the results are then divided by 100 so they can be compared to the other results

TABLE 6.4: The benchmark problem instances used in this chapter

Instance set name	Number of Instances	Number of Rectangles	Sheet Width	Optimal Height
Hopper and Turton (2001)	21	16-197	20-160	20-240
Valenzuela and Wang (2001)	12	25-1000	100	100
Burke, Kendall and Whitwell (2006)	12	10-500	40-100	40-300
Ramesh Babu and Ramesh Babu (1999)	1	50	1000	375

in the literature. For each instance modified in this way, the total area of all the shapes divided by the width of the sheet is never less than 100, so we can say that this procedure never results in an easier instance, and it is fair to compare the results with others in the literature.

Fitness Function

The individual's fitness is the height of the solution that it creates. Lower fitnesses are therefore better, because lower solutions mean that the pieces are better packed. Individuals are selected to be parents of the next generation by tournament selection, selecting seven individuals at random, and then selecting the individual with the best fitness from the seven. When two heuristics have the same fitness, the winner is the heuristic which wastes the least space between the pieces. This does not include wasted space at the top of the solution, only wasted space that is completely enclosed. There is therefore selection pressure to pack the pieces without gaps, fitting the pieces together neatly.

6.4 Results

6.4.1 Explanation of Benchmark Algorithms

Table 6.5 shows the overall results we have obtained for the system. We compare our results to two metaheuristic methods described in section 2.4.4 of the literature review, a genetic algorithm with bottom-left-fill decoder (GA+BLF) [143], and a simulated annealing approach with bottom-left-fill decoder (SA+BLF) [143]. The metaheuristics search the space of potential piece orderings, and the bottom-left-fill places each piece in turn into the lowest

TABLE 6.5: Results of our evolved heuristics compared to recent metaheuristic and constructive heuristic approaches, and the reactive GRASP approach. The results are highlighted, in bold type, where the hyper-heuristic system finds a heuristic which has equal or better results than the metaheuristic approaches and the best-fit algorithm for that instance. The GRASP algorithm is generally the best approach, the bold values are intended to show how the evolved heuristics compare to the other approaches

Name	Meta-heuristic	BF Heuristic	Evolved 64	Evolved 128	Evolved 256	Reactive GRASP	Max Time to Evolve (s)
N1	40	45	**40**	**40**	**40**	40	0.141
N2	51	53	52	**51**	52	50	5.27×10^1
N3	52	52	**52**	**51**	**51**	51	2.06×10^1
N4	83	83	**83**	**81**	**82**	81	3.96×10^2
N5	106	105	**103**	**103**	**105**	102	1.80×10^2
N6	103	103	**102**	**102**	**101**	101	6.21×10^2
N7	106	107	**102**	**102**	**102**	101	1.19×10^3
N8	85	84	**82**	**83**	**82**	81	1.52×10^3
N9	155	152	154	**151**	**152**	151	1.67×10^3
N10	154	152	**152**	**152**	**152**	151	9.48×10^3
N11	155	152	**152**	**152**	**151**	151	1.19×10^4
N12	312	306	**305**	311	**304**	303	7.55×10^4
c1p1	**20**	21	21	21	21	20	2.08×10^1
c1p2	21	22	**21**	**21**	**21**	20	3.38×10^1
c1p3	20	24	21	**20**	**20**	20	7.38
c2p1	16	16	**16**	**16**	**16**	15	6.73×10^1
c2p2	16	16	**16**	**16**	**16**	15	1.60×10^1
c2p3	16	16	**16**	**15**	**16**	15	1.63×10^1
c3p1	32	32	**31**	**31**	**31**	30	5.08×10^1
c3p2	32	34	**32**	**32**	**32**	31	1.75×10^2
c3p3	32	33	34	**32**	34	30	4.42×10^1
c4p1	64	63	64	**63**	**62**	61	2.34×10^2
c4p2	63	62	**61**	**62**	**62**	61	2.78×10^2
c4p3	62	62	**61**	**62**	**62**	61	1.25×10^2
c5p1	94	93	**93**	**91**	**92**	91	1.24×10^3
c5p2	95	92	93	**91**	**92**	91	7.21×10^2
c5p3	95	93	**91**	**93**	**91**	91	1.48×10^3
c6p1	127	123	**122**	**123**	**121**	121	2.65×10^3
c6p2	126	122	132	**122**	**121**	121	2.95×10^3
c6p3	126	124	**122**	**123**	**122**	121	1.72×10^3
c7p1	255	247	**245**	**245**	**245**	244	2.32×10^3
c7p2	251	244	**243**	246	**243**	242	4.83×10^3
c7p3	254	245	246	**243**	**243**	243	2.65×10^4
NiceP1	108.2	107.4	**106.73**	107.05	**106.56**	103.7	1.41×10^2
NiceP2	112	108.5	**107.17**	108.76	**107.97**	104.6	9.67×10^2
NiceP3	113	107	**104.84**	110.5	**105.58**	104	1.40×10^3
NiceP4	113.2	105.3	120.85	118.82	**104.18**	103.6	1.05×10^4
NiceP5	111.9	103.5	110	**102.6**	**101.97**	102.2	8.13×10^4
NiceP6	-	103.7	**103.1**	**101.69**	**101.95**	102.2	2.81×10^5
PathP1	106.7	110.1	110.61	112.86	**108.3**	104.2	1.03×10^2
PathP2	107	113.8	108.98	108.73	**104.53**	101.8	7.37×10^2
PathP3	109	107.3	126.21	113.08	**106.87**	102.6	2.37×10^3
PathP4	108.8	104.1	109.27	106.54	104.95	102	6.69×10^3
PathP5	111.11	103.7	114.05	121.97	106.18	103.1	1.34×10^5
PathP6	-	102.8	126.5	118.81	115.45	102.5	6.90×10^5
RBP1	400	400	**400**	**400**	**400**	375	5.40×10^2

available position, including any 'holes' in the solution, and then left-justifying it. Table 6.5 shows only the best result of the two on each instance (shown in the "metaheuristic" column). These two metaheuristic approaches are also described in [143], and the results evaluated using a packing density measure rather than the length of sheet needed. To obtain the results that we compare with here, the metaheuristic methods were implemented again in [38].

The 'best-fit' style algorithm is presented in [38], and has achieved superior results to BL and BLF (see section 2.4.3 for an explanation of BL and BLF). The best-fit algorithm is utilised here as a constructive, non metaheuristic, benchmark. The operation of the best-fit heuristic is similar to the framework within which the evolved heuristics operate. For example, the pieces are packed one at a time, but any piece can be chosen from all of those available, and best-fit is also deterministic. The best-fit heuristic must pack the piece into the lowest available bin, and always into either the left or right corner (this is a user defined parameter). In contrast, the heuristics in this chapter evolve the choice of bin and location. The results are highlighted, in bold type, where the hyper-heuristic system finds a heuristic which has equal results or beats the metaheuristic approaches and the best-fit algorithm for that instance.

We also compare with the reactive GRASP presented in [4]. The reactive GRASP method does not allow piece rotations, but they are allowed for the heuristics evolved in these experiments. The GRASP results would probably not be worse if rotations *were* allowed, so while we are aware of the difference, we believe the comparison with GRASP is still valuable, as it is a comparison with a complex human designed heuristic with many tunable parameters.

Computational times for the genetic programming runs are shown in the far right column of table 6.5. The figure reported is for the longest of the three runs (with 64, 128, and 256 population), which in most cases was the run with the population set to 256. The run times are relatively large compared to the run times reported for the methods that we compare against. It is important to keep in mind that this includes the time taken for the genetic programming to both design and run a heuristic. The times stated in the literature for the GRASP, metaheuristics, and the best-fit heuristic, only include the time taken to run them. We assert that a fair comparison would involve the time taken for the human designers to build, as well as run, their heuristics. However, the contribution of these results is not to show that this methodology is quicker, but to show that a heuristic design process

can be automated for two dimensional packing.

6.4.2 Comparison with Metaheuristics and the Best-Fit Heuristic

Table 6.5 shows that in the vast majority of cases, the GP system finds a heuristic in at least one of the three runs that obtains packings better than, or equal to, the two established metaheuristics and the best-fit algorithm. However, there are four exceptions. For the instance c1p1, the GP system cannot find a heuristic that performs better than either the GA+BLF or the SA+BLF, both of which find the optimal solution at a height of 20. It is known in the literature that the metaheuristic approaches perform well on problems of small size because the search space is relatively small. Also, no heuristic is found that is better than the best-fit heuristic on the three most difficult 'Path' problems from [267]. Both of the metaheuristics were unable to find a solution to the NiceP6 and PathP6 problems due to excessive time requirements, so it would seem that the simpler constructive heuristic performs best for these larger problems.

6.4.3 Comparison with Reactive GRASP

The evolved heuristics' results are competitive with the reactive GRASP method [4] explained in section 2.4.4 of the literature review. On the majority of problems, the reactive GRASP obtains results which are the same or better than the evolved heuristics. Table 6.6 presents the mean and standard deviation of the differences between the reactive GRASP results and the best evolved heuristic for each problem. The table shows these both as percentages and actual units, and it shows these calculations on all the results and on the three individual benchmark sets which contain more than one instance.

Table 6.6 therefore summarises the observation that on both the Hopper and Turton, and Burke, Kendall and Whitwell datasets, the best evolved heuristics for those problems match the reactive GRASP or are one unit worse, apart from the problem c3p3, where the best evolved heuristic is two units worse. However, on each of the problems NiceP5 and NiceP6, at least one heuristic is generated that beats the result of the reactive GRASP, which is a significant result on two relatively large problems. This is especially true as the evolved heuristics are only constructive, with no post-processing such as an iterative improvement stage.

TABLE 6.6: Difference in performance of the best evolved heuristics for each instance and the reactive GRASP, the figures represent the extent to which the GRASP performs better than the evolved heuristics.

Instance set name	Percent		Units	
	Mean	S.D.	Mean	S.D.
All Instance Sets	1.91%	2.69%	1.72	4.09
Burke, Kendall and Whitwell (2006)	0.52%	0.66%	0.50	0.52
Hopper and Turton (2001)	1.90%	2.58%	0.57	0.60
Valenzuela and Wang (2001)	2.93%	3.42%	3.02	3.51

6.4.4 Example Packings

Figures 6.13 and 6.14 show two packings achieved for the problem instance c5p3, both of which have a height of 91, just one unit over the optimal packing. Figures 6.15 and 6.16 show the two evolved heuristics that created these packings, where W and H are the piece width and piece height respectively. OBH and NBH are the opposite bin height and the neighbouring bin height. BWL is the bin width left, and BH is the bin height. All of these genetic programming terminals are summarised in table 6.1.

The heuristics have a tree structure, but could not be displayed in this way because of space constraints, and so they are displayed in polish notation, with the operator as a prefix. These two heuristics are not easily understood, and it is likely that they contain redundancy as is common in the output of a genetic programming run. They are offered not as an explanation of why the pieces were packed in this way, but as examples of the output of the hyper-heuristic. They illustrate that the evolved heuristics are complex, and take a different form to a heuristic that is developed by a human programmer.

Figure 6.15 was evolved when the population was set to 64, and figure 6.16 was evolved when the population was set to 256. Figure 6.15 is a heuristic that packs the pieces of this instance leaving no gaps at all, as figure 6.13 shows. Interestingly, figure 6.16 is a heuristic that decides to pack one of the smallest pieces first, seen in the lower-left corner of figure 6.14. Even though this may seem counter-intuitive to humans, it is a strategy that pays off in the long term for this problem instance.

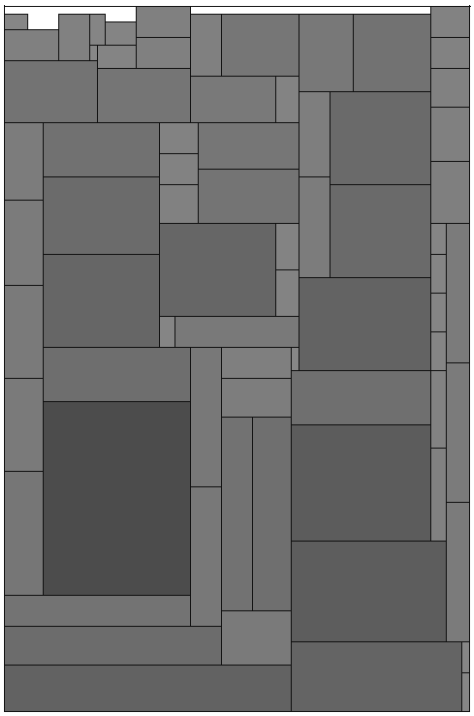

FIGURE 6.13: Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 64

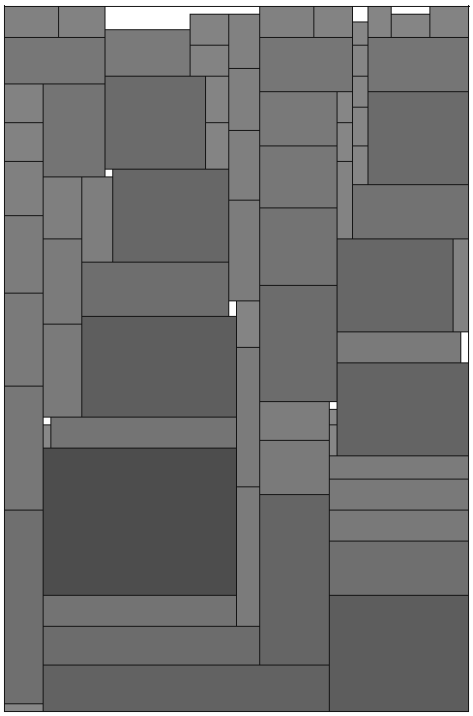

FIGURE 6.14: Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 256

```
(- (+ (- (+ (* NBH (- W W)) (- (- (+
(* NBH (* (- W W) (- W W))) (- (-
(* (+ BH BWL) (- W W)) NBH) (*
BH BH))) (- (+ (% H OBH) (+ W
NBH)) (- (* W W) (* BH BH)))) (+
(* BH BH) (- (+ BH BWL) OBH))))
(- (+ (* BH BH) (- (+ BH BWL)
OBH)) (- W (* BH BH)))) (- (* (+
BH BWL) (- W W)) (+ BH BWL)))
(- (+ (+ W BH) (+ W NBH)) (- (*
W W) (* W W))))
```

FIGURE 6.15: The heuristic that packed figure 6.13

```
(- (- (- (- (- (- (+ OBH W) (- BH
W)) (- BH W)) (% (+ NBH OBH)
(+ NBH BH))) (% (+ NBH OBH)
(% BWL BWL))) (% (+ OBH W)
(% (- (- (- (- (+ OBH W) (- BH
W)) (- (% (+ NBH OBH) (- (- (-
(- BH (- NBH BWL)) (- W W)) (%
(% (+ NBH OBH) (% BWL BWL))
(+ NBH BH))) BH)) W)) (- BH (-
NBH BWL))) (- BH W)) NBH)))
BH)
```

FIGURE 6.16: The heuristic that packed figure 6.14

6.4.5 Results with Parsimony Pressure and Cache of Heuristic Evaluations

It has been shown that the process of evolving a heuristic for an instance produces results which are competitive with those of human-created heuristics. The problem is that the process of evolution often takes a long time. As time can be an important factor, the experiments were repeated, with two measures taken to attempt to reduce the time of the run.

The first is a method to control code bloat, which is a phenomenon in genetic programming [228, 202, 253, 176] where the average size of the trees in the population grows as the number of generations increase. This can cause a problem if left unchecked, because the time taken to evaluate the individual increases with its size. This is especially true in the experiments here, because the heuristic is evaluated a large number of times in order to pack an instance. The method employed to reduce the impact of code bloat in the experiments is the tarpeian wrapper method of Poli [226]. A proportion of the individuals with above average size receive the worst fitness value possible. The proportion is a parameter of the method, set to 0.2 in these experiments.

The second addition to the implementation is a memory, for caching the results of the heuristic evaluations. When a piece is placed into the solution, the heuristic is then evaluated again for all the allocations available. This results in wasted time, because the value that the heuristic returns for certain allocations will be the same as it was at the last decision point. The only values that change are for the allocations affected by the newest piece. The allocations associated with the two new bins created by the piece will need to be evaluated by the heuristic for all the pieces. Also, the two bins on either side of the bin which received the piece will need to be re-evaluated, because they are involved in the calculation of the OBH and NBH terminals. All the allocations associated with all the other bins will receive the same value from the heuristic, because the values of the heuristic's terminals will acquire the same values.

The improved implementation, therefore, provides each bin with a memory, storing the heuristic's value for each piece and orientation in that bin. A bin's memory is deleted when the heuristic decides to put the piece into it, or an one adjacent to it. The heuristic is therefore only actually evaluated when there is a chance that the value returned by it could be different. This results in less time per run for the larger instances, because for those instances there are more bins created on the sheet and therefore more allocations which

TABLE 6.7: A comparison of the time taken to evolve the heuristics using the previous implementation, and the improved implementation.

	Previous Implementation		Improved Implementation	
Instance Name	Best Result	Max Time to Evolve (s)	Best Result	Max Time to Evolve (s)
N1	40	0.1	40	0.2
N2	51	52.7	51	59.2
N3	51	206.1	51	150.0
N4	81	395.7	81	369.6
N5	103	180.1	103	436.8
N6	101	620.9	101	650.1
N7	102	1191.9	103	727.4
N8	82	1521.8	81	958.2
N9	151	1665.2	152	910.6
N10	152	9483.1	151	6324.4
N11	151	11905.4	151	21427.4
N12	304	75487.4	302	51929.2
c1p1	21	20.8	21	27.2
c1p2	21	33.8	21	32.8
c1p3	20	7.4	20	11.5
c2p1	16	67.3	16	99.1
c2p2	16	16.0	16	93.4
c2p3	15	16.3	15	70.0
c3p1	31	50.8	31	81.7
c3p2	32	174.6	32	107.2
c3p3	32	44.2	31	104.4
c4p1	62	233.8	61	467.6
c4p2	61	278.1	62	371.4
c4p3	61	125.3	61	256.5
c5p1	91	1236.1	91	683.1
c5p2	91	720.8	91	762.4
c5p3	91	1476.2	92	489.2
c6p1	121	2646.3	122	1240.1
c6p2	121	2949.9	121	1502.2
c6p3	122	1715.3	122	1535.4
c7p1	245	2317.9	241	8152.4
c7p2	243	4831.9	243	4367.6
c7p3	243	26500.7	243	5205.0
NiceP1	106.56	141.5	106.74	80.7
NiceP2	107.17	966.8	105.85	358.7
NiceP3	104.84	1401.3	103.79	1588.2
NiceP4	104.18	10495.6	101.81	12399.3
NiceP5	101.97	81301.9	101.68	69069.0
NiceP6	101.69	280586.3	N/A	N/A
PathP1	108.30	102.5	106.80	101.7
PathP2	104.53	737.3	103.73	373.9
PathP3	106.87	2373.6	103.55	1909.4
PathP4	104.95	6692.8	107.74	7010.4
PathP5	106.18	133852.1	109.35	57709.1
PathP6	115.45	690178.7	N/A	N/A
RBP1	400	539.8	400	308.8

are evaluated needlessly. The new implementation also uses floating point data rather than integer data to represent the bins, so the Valenzuela and Wang data set no longer needs to be converted to integers.

Table 6.7 shows the results of the improved implementation compared to the previous implementation. The largest problem instances are NiceP6 and PathP6, each containing 1000 pieces with floating point dimensions. No result is reported for these because the new implementation causes the program to run out of memory due to the high number of bins that are generated for these instances. The 'best result' column shows the best of the three runs (population 64, 128, and 256), and the 'max time to evolve' column shows the time taken for the longest of the three runs, which in most cases is the run with the population set to 256. The table shows that the evolved heuristics are of approximately the same quality. However, the time taken to evolve the heuristics is reduced. This effect becomes more evident as the size of the problem instance increases.

In summary, the results of table 6.7 show that it is possible to reduce the time it takes to evolve a heuristic by implementing the processes more efficiently.

6.5 Conclusion

This chapter has presented a framework and representation that enables heuristics to be automatically generated for two dimensional strip packing. The heuristics construct a solution one piece at a time, by choosing a piece to pack next and where to place it. The selection is made by returning a value for each available allocation, and performing the allocation which receives the highest value. Each heuristic is evolved to exploit features and properties of one instance.

In contrast to the heuristics evolved for one dimensional bin packing in chapter 4 and 5, the heuristics are disposable, in the sense that they are not applied to unseen instances after they have been evolved. The heuristics *can* be applied to unseen instances, and they would generate legal solutions. However, the solutions would not be as close to optimal as the results on their training instance, because the instances are too different for the heuristics to be able to exploit any similarities. Future work could involve generating data sets which show that these heuristics can be re-usable under certain conditions.

The results obtained by the evolved heuristics are better than the results of human created metaheuristics, and the best available constructive heuristic. One reason for

the results being better could be because the hyper-heuristic methodology automatically generates a constructive heuristic for each instance, meaning that the result for an instance is obtained by a heuristic which is tuned to that instance.

The reactive GRASP methodology [4] for this problem is a complex method with many parameters, and both constructive and iterative phases. The results show that the GRASP methodology produces superior results to those of the evolved heuristics. An explanation for this could be that the set of functions and terminals are not expressive enough to enable the genetic programming to easily find heuristics which can obtain the same quality of results. Compared directly to the hyper-heuristic methodology in this chapter, the results are better and it obtains them in a quicker time. However, it must be noted that the hyper-heuristic methodology takes a long time to run because it is designing *and* running the heuristics, whereas the time reported for GRASP is only to run it.

The contribution of this work is not to obtain the best results in the literature for these instances, but to investigate methods to automate the heuristic design process. These results show that heuristics *can* be automatically designed for this problem, and that they are of a high enough quality to warrant future investigation. If it can be discovered that the heuristics can be reused under certain circumstances, the large run times can then be further justified, because the heuristics are very quick to return a solution to new problems once they are evolved, due to their constructive nature.

Reducing run times would still be beneficial, and this could be achieved by focussing further work on evolving heuristics which intelligently select which allocations to consider, rather than blindly evaluating all those available. This has the potential to reduce the time taken to evolve a heuristic, because it is the many evaluations of the heuristics in the population that account for the large run times. However, it is likely that the run times will always be relatively large compared to other work in the literature, because this methodology must design and run the heuristics, whereas other methodologies only take into account the time taken to run. It is the design stage which is the most time intensive, and this is exactly why it is potentially beneficial to research ways of automating the process.

The next chapter presents a hyper-heuristic system which can evolve heuristics for any one, two or three dimensional bin packing or knapsack problems. This is the first such system which can claim to obtain results for each of these problems with no change in parameters between runs. This represents further work towards one of the objectives of

hyper-heuristic research and, indeed, this thesis, to raise the level of generality at which optimisation systems can operate.

Chapter 7

A Hyper-Heuristic for 1D, 2D and 3D Packing Problems

7.1 Introduction

Through chapters 3 to 6, the problem domains for which heuristics have been evolved have progressed from one to two dimensions. This naturally leads to the question of whether this methodology can also be applied to three dimensional problems. Answering this question is one of the aims of this chapter. However, the investigation goes further than this. The hyper-heuristic system presented in this chapter can evolve heuristics for one, two, or three dimensional bin packing or knapsack problems. No current packing methodology is able to produce solutions for such a wide range of packing problems. The hyper-heuristic system also does not require any tuning of parameters between obtaining a solution for a one two or three dimensional problem, or when switching between a bin packing and knapsack problem. This is possible because the hyper-heuristic searches the solution space indirectly, by searching the heuristic space instead.

It could be argued that, for example, genetic algorithms have already been applied to all three domains [96, 146, 129, 187, 116]. However, in each of these cases, the genetic algorithm has been implemented in a different way, with a different representation and different parameters. Even though the genetic algorithm *paradigm* has been used for both problem domains in all three dimensions, the implementation of the actual algorithms are quite different. To the author's knowledge, the hyper-heuristic system presented in this chapter is the first algorithm to be able to obtain results on all of these problem domains, and to do so without requiring any of the parameters to be changed. It has a common

problem representation, and a set of parameters that need not change, because they apply to the heuristics rather than the underlying problem.

Increasing the dimensionality of the problem to three dimensions presents further challenges in designing a suitable representation, with which the heuristics can build a solution. In one dimension, a piece can be placed into only one location in a bin. In two dimensions, there are many locations where a piece can be placed, including both a lateral position and a height. The heuristic must also decide which orientation the piece is to have, whereas the one dimensional packing problem does not require such a decision. Pieces in three dimensional packing problems have six possible orientations, and the depth dimension adds much more complexity to the problem of finding a suitable location. The representation explained in this chapter is designed with these issues in mind.

In chapter 6, the container was represented as a number of dynamic bins, which determined the potential locations for a piece. In the experiments of this chapter, the container is represented by a number of 'corners', which represent locations, and contain information about their immediate surroundings. The two dimensional problem representation started with one bin, representing the whole container. Similarly, the three dimensional representation starts with one corner, into which the first piece is placed. When a piece is put into the corner it creates three more, and further corners are created as more pieces are put into the container.

This somewhat limits the potential places where a piece can go, and it essentially means that each piece must be placed into a location where it is adjacent to an existing piece. However, the decision was taken because of the length of time it took to evolve heuristics in the previous chapter. As there are even more potential combinations of location and orientation for every piece, the locations must be limited in order to keep the run times to acceptable levels.

The outline of this chapter is as follows. The formulations for the two problem domains are stated in section 7.2. Section 7.3 explains the representation that the heuristics manipulate in order to construct a solution. In section 7.4, the methodology is explained, including how a heuristic packs a problem instance, and how the heuristics are evolved. The data sets are described in section 7.5. They are in a separate section in this chapter because there are 18 in total, which is many more than in previous chapters. The results obtained on these data sets are presented in section 7.6, and the conclusions are drawn in section 7.7.

7.2 Problem Descriptions

The descriptions of the problems addressed in this thesis were given in the literature review (chapter 2). For ease of reference, this section re-states the formulations for the bin packing and knapsack problems that are addressed in this chapter.

7.2.1 The Knapsack Problem

The one dimensional 0-1 knapsack problem consists of a set of pieces, a subset of which must be packed into one knapsack with capacity c. Each piece j has a weight w_j and a value v_j. The objective is to maximise the value of the pieces chosen to be packed into the knapsack, without the total weight exceeding the capacity of the knapsack. A mathematical formulation of the 0-1 knapsack problem is shown in equation 7.1, taken from [199].

$$\begin{aligned} \text{Maximise} \quad & \sum_{j=1}^{n} v_j x_j \\ \text{Subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j \in \{0,1\}, \qquad\qquad j \in N = \{1,\ldots,n\}, \end{aligned} \tag{7.1}$$

Where x_j is a binary variable indicating whether piece j is selected to be packed into the knapsack.

7.2.2 Knapsack Packing in Two and Three Dimensions

The knapsack problem can be defined in two dimensions. In the two dimensional case, the knapsack has a width W and a height H. Each piece $j \in N = \{1,\ldots,n\}$ is defined by a width w_j, a height h_j, and a value v_j. The objective is to maximise the total value of the pieces chosen to be packed into the knapsack. No piece can overlap another, or exceed the boundaries of the knapsack.

The knapsack problem can also be defined in three dimensions, with the knapsack having a depth D, and each piece having a depth d_j, in addition to the properties in the two dimensional case. In both the two and three dimensional cases we allow all rotations of the pieces where the sides of the piece are parallel to the edges of the knapsack. We also do not impose the 'guillotine' cutting constraint defined in [63].

7.2.3 The Bin Packing Problem

The classical one dimensional bin packing problem consists of a set of pieces, which must be packed into n bins. Each piece j has a weight w_j, and each bin has capacity c. The objective is to minimise the number of bins used, where each piece is assigned to one bin only, and the weight of the pieces in each bin does not exceed c. A mathematical formulation of the bin packing problem is shown in equation 7.2, taken from [199].

$$\begin{aligned}
\text{Minimise} \quad & \sum_{i=1}^{n} y_i \\
\text{Subject to} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, && i \in N = \{1, \ldots, n\}, \\
& \sum_{i=1}^{n} x_{ij} = 1, && j \in N, \\
& y_i \in \{0, 1\}, && i \in N, \\
& x_{ij} \in \{0, 1\}, && i \in N, j \in N, \qquad (7.2)
\end{aligned}$$

Where y_i is a binary variable indicating whether bin i has been used, and x_{ij} indicates whether piece j is packed into bin i.

7.2.4 Bin Packing in Two and Three Dimensions

The bin packing problem can be defined in two dimensions, where the bin size is defined by a width W and a height H. Each piece $j \in N = \{1, \ldots, n\}$ is defined by a width w_j and a height h_j. The objective is to minimise the number of bins needed to accommodate all of the items, where each item is assigned to one bin only. The pieces must not overlap each other or exceed the boundaries of their bin.

The problem is often referred to as the two dimensional stock cutting problem. This is because of its similarities with the real world problem of cutting shapes from standard sized stock sheets of material, while attempting to minimise the number of sheets used. In the experiments of this chapter, we allow 90° rotations of the pieces, and do not impose the 'guillotine' constraint defined in [63].

The bin packing problem can also be defined in three dimensions, where the bins also have a depth D, and each piece $j \in N = \{1, \ldots, n\}$ also has a depth d_j, in addition to the properties in the two dimensional case. This problem is often referred to as the multiple

container loading problem because of its similarities with real world problems encountered in industries such as shipping and home delivery, where reducing the number of vehicles needed will reduce costs. In the experiments of this chapter, we allow orthogonal rotations of the pieces in all directions, except where the instance itself specifies that only certain rotations are allowed for each piece.

7.3 Representation

We will begin the description of our representation by using the example of a three dimensional knapsack problem. We will then extend the description in section 7.3.5 to cover the bin packing problem, and to cover problems of lower dimensions.

7.3.1 Bin and Corner Objects

Each bin is represented by its dimensions, and by a list of 'corner' objects, which represent the available spaces into which a piece can be placed. A bin is initialised by creating a corner which represents the lower-back-left corner of the bin, as shown in figure 7.1. Therefore at the start of the packing, the heuristic just chooses which piece to put in this corner, because it is the only corner available. Figure 7.1 also shows the positive directions of the X, Y, and Z axes.

When the chosen piece is placed, the corner is deleted, as it is no longer available, and at most three more corners are generated in the X, Y and Z directions, from the coordinates of the deleted corner. This is shown in figure 7.2. A corner is not created when the piece meets the outside edge of the container. Therefore, after the first piece has been put into the bin, the heuristic then has a choice of a maximum of three corners that the first piece defines.

A corner contains information about the three 2D surfaces that define it, in the XY plane, the XZ plane, and the YZ plane. The corner is at the intersection of these three orthogonal planes. Figure 7.3 shows the three 2D surfaces that the corner above the piece is defined by. Note that the XZ surface has its limits at the edges of the top of the piece. Similarly, figure 7.4 shows the three surfaces of the corner that is to the right of the piece in the figure.

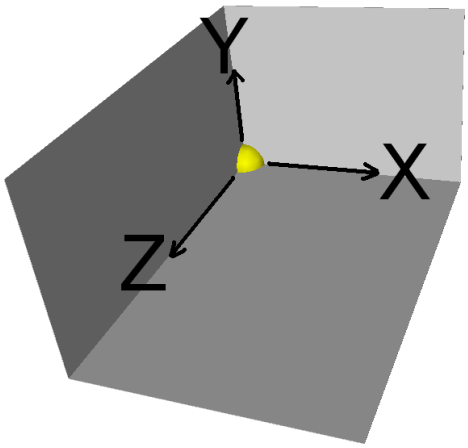

FIGURE 7.1: An initialised bin with one 'corner' in the back-left-bottom corner of the bin

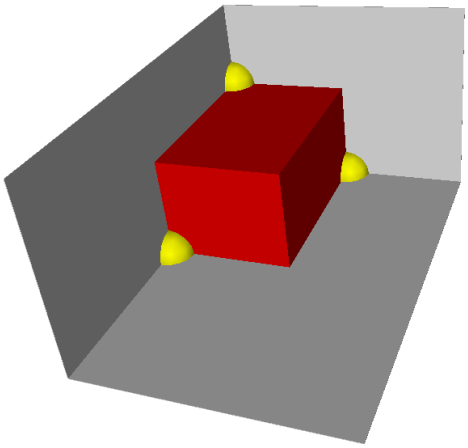

FIGURE 7.2: A bin with one piece placed in the corner from figure 7.1

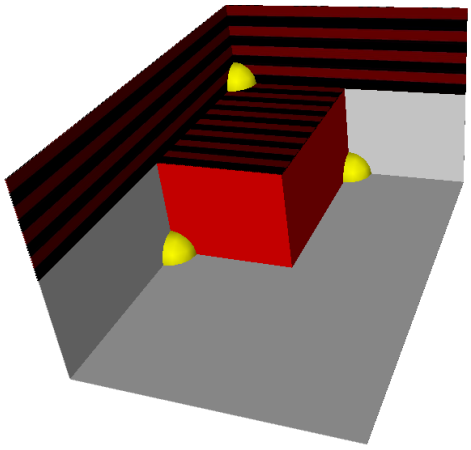

FIGURE 7.3: The three surfaces defined by the corner that is in the Y direction of the piece

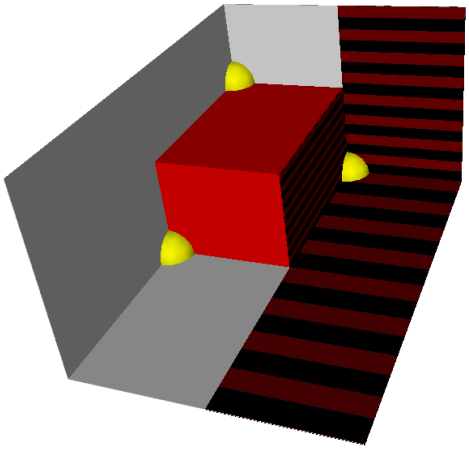

FIGURE 7.4: The three surfaces defined by the corner that is in the X direction of the piece

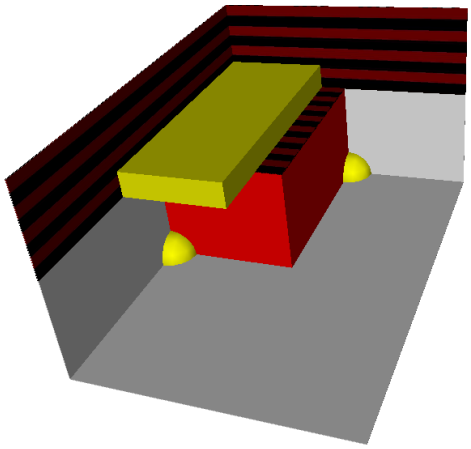

FIGURE 7.5: An invalid placement of a new piece, because it exceeds the limit of the corner's XZ surface in the Z direction

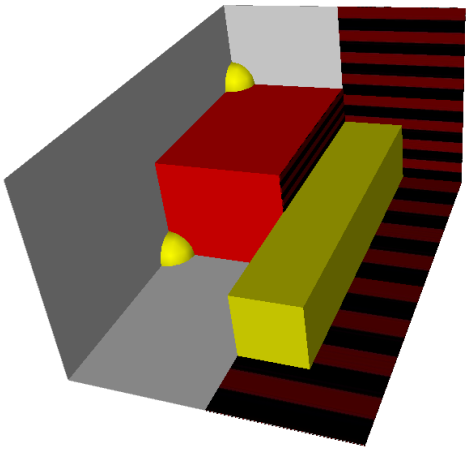

FIGURE 7.6: An invalid placement of a new piece, because it exceeds the limit of the corner's YZ surface in the Z direction

7.3.2 Valid Placement of Pieces

Each piece is considered by the heuristic in all six orientations at every corner, unless the instance itself constrains the orientation of the piece. Only an orientation that will fit against all three surfaces of the corner without exceeding any of them, is considered to be a valid orientation. Figure 7.5 shows an invalid placement, because a new piece exceeds the limit of the corner's XZ surface in the Z direction. Also, in figure 7.6, the new piece exceeds the limit of the corner's YZ surface in the Z direction, so this placement is invalid.

7.3.3 Extending a Corner's Surfaces

If a piece is put into a corner and the piece reaches the limit of one of the corner's surfaces, it often means that a surface of one or more nearby corners needs to be modified. An example is shown in figure 7.7, where the piece is placed in the middle and the two corners shown must have their surfaces updated. In this situation, as is often the case, the piece does not extend an existing surface, but creates a new one in the same plane that overlaps the existing surfaces. So the corner on the left of figure 7.7 now has two surfaces in its XZ plane, and the corner on the right has two surfaces in its YZ plane.

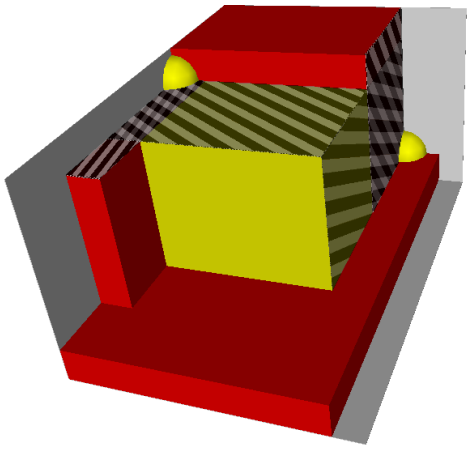

FIGURE 7.7: A piece which reaches the limit of two surfaces of the corner that it was put into

7.3.4 Filler Pieces

Some corners may have surfaces which are too small, in one or more directions, for any piece. If this is the case then the corners are essentially wasted space. If left unchecked, eventually there will be many corners of this type that no piece can fit into, and there may be potentially a lot of wasted space that could be filled if the pieces were allowed to exceed the limits of the three surfaces of a corner. For this reason, we use 'filler pieces' that effectively fill cuboids of unusable space. As we will describe in this section, they create more space at one or more nearby corners by extending one of their surfaces, so that more pieces can potentially fit into them.

After a piece has been placed, the corner with the smallest available area for a piece is checked to see if it can accommodate any of the remaining pieces. If it cannot, then we put a filler piece in this corner. The filler piece will have dimensions equal to the closest limits of the corner's surfaces in each direction, so the filler piece does not exceed any of the limits of the three surfaces of a corner. In figure 7.8, the three surfaces are shown of the corner with the smallest available area. If no remaining piece can fit into this corner then it is selected to receive a filler piece, which is shown in figure 7.9 after it is placed.

The reason for the filler piece is that it will exactly reach the edge of an adjacent piece, and extend the surface that it matches with, therefore increasing the number of pieces that will fit into the corner that the surface belongs to. As is shown in figure 7.9, three

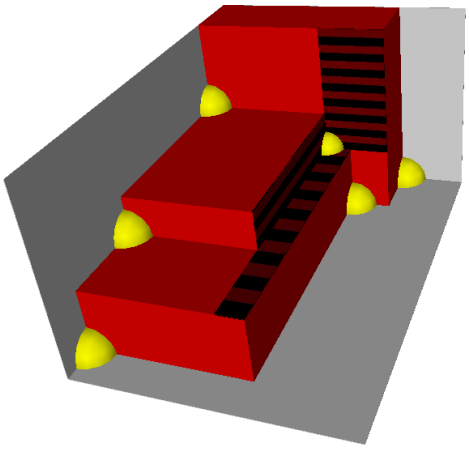

FIGURE 7.8: The three surfaces of the corner with the smallest available area

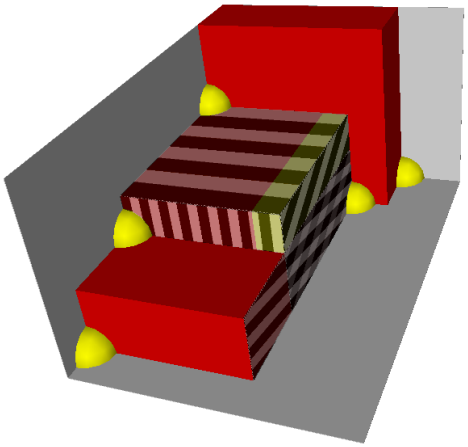

FIGURE 7.9: A filler piece is put into the corner from 7.8, the surfaces of two corners are updated

corners have their surfaces updated by the filler piece. First, the corner in the top left of the figure has its XZ surface extended across the top of the filler piece in the X direction, shown by horizontal stripes. Secondly, the corner to the left of the filler piece has its XY surface extended across the front of the piece, shown by vertical stripes. Thirdly, the corner just under the filler piece receives a second YZ surface, which extends up the right side of the filler piece. Both the original YZ surface and the second one are shown in the figure, as diagonal stripes. Note that a filler piece never creates a corner, the net effect of inserting a filler piece will be the deletion of the corner that the filler piece was inserted into.

After the filler piece has been placed, this process is repeated, and if the corner with the next smallest available area cannot accept any remaining piece then that corner is filled with a filler piece. When the smallest corner can accept at least one piece from those remaining, the packing continues. In the knapsack problem there is only one bin. The packing will terminate when the whole bin is filled with pieces and filler pieces, and there are no corners left. The result is then the total value of the pieces that have been packed.

7.3.5 Bin Packing, and Packing in Lower Dimensions

The representation described in sections 7.3.1-7.3.4 also allows for the bin packing problem to be represented. If the user specifies that the instance is to be used for the bin packing problem, then an empty bin is always kept open, so the evolved heuristic can choose to put

a piece into it. Then, when this bin is first used, another new bin is opened. The heuristic always has the choice of any corner in any bin. When running an instance as a bin packing problem, the piece values (necessary for the knapsack problem) are set to one because they are not relevant, and the packing stops when all the pieces have been packed.

The three dimensional representation can also be used for two and one dimensional problem instances, by setting the redundant dimensions of the bins and pieces to one. For example, when using a two dimensional instance, the depth of each piece and bin is set to one, and for a one dimensional instance, the depth and height are set to one.

7.4 Methodology

This section describes the hyper-heuristic algorithm that evolves the packing heuristics. We will refer to a combination of piece, orientation and corner as an 'allocation', and a point in the algorithm where the heuristic is asked to decide which piece is placed where will be referred to as a 'decision point'. Section 7.4.1 describes how a heuristic is applied to a problem to obtain a solution, explaining what the heuristics choose at every decision point. Section 7.4.2 explains the functions and terminals of the heuristics evolved for this chapter. Section 7.4.3 describes how the heuristics themselves are evolved, including the genetic programming parameters and the fitness function.

7.4.1 How the Heuristic is Applied

The heuristic operates within a fixed framework which, at each decision point, evaluates the heuristic once for each valid allocation that can be performed. So every piece, orientation and corner combination, in every bin, is evaluated in this way. The actual allocation performed is the one which receives the maximum evaluation. Then the filler stage (section 7.3.4) is performed to fill any redundant space, and the cycle then repeats by considering the next decision point.

The framework is shown in algorithm 6. The while loop on line 1 performs as many iterations as is necessary to pack all of the pieces. There may be more iterations of this loop than there are numbers of pieces, because of the 'if' statement on line 2. This checks if the corner with the smallest area can accommodate a piece, and if not then it is filled with a filler piece. If a filler piece *is* placed into the corner, then the main block of code is not performed due to the 'else' statement on line 4, and the next smallest corner is

Algorithm 6 Pseudo code of the framework within which the heuristics are applied in chapter 7

```
1: while pieces exist to be packed do
2:   if no piece can fit into the corner with the smallest area then
3:     put a filler piece into this corner and update the corner structure
4:   else
5:     for all pieces P in pieceList do
6:       for all orientations O of the piece do
7:         for all corners C in current partial solution do
8:           if piece P in orientation O fits into corner C then
9:             current_allocation = P,C,O
10:            heuristic_output = evaluateHeuristic(current_allocation)
11:            if heuristic_output > best_output then
12:              best_allocation = current_allocation
13:              best_output = heuristic_output
14:            end if
15:          end if
16:        end for
17:      end for
18:    end for
19:    perform best_allocation on partial solution
20:    remove chosen piece from pieceList
21:    update corner structure
22:    if solving a bin packing problem AND the chosen corner was in an empty bin then
23:      open a new bin containing one available corner
24:    end if
25:    if there are no corners left then
26:      the knapsack packing is complete, break from while loop
27:    end if
28:  end if
29: end while
```

checked on the next iteration of the while loop.

If at least one piece can fit into the corner with the smallest area, then we are sure that one piece can be legally packed, and so the main block of code executes (lines 5 to 27). There are three nested 'for' loops, on lines 5, 6 and 7, which ensure that all of the piece, orientation and corner combinations are considered by the heuristic. If the piece can physically fit into the corner, in its current orientation, then the heuristic is evaluated using the current allocation as input. This is shown on line 10 of the algorithm. This means that its terminal values are set to values dictated by the properties of the current allocation, i.e. the current piece, orientation and corner (see section 7.4.2 for further clarification of this process). One numerical value is returned by the heuristic, which is interpreted as an evaluation of the relative suitability of the allocation. If this evaluation is the highest returned so far for any allocation, then it is stored as the best allocation so far (line 12 of the algorithm).

When all of the possible allocations have been considered by the heuristic, the algorithm progresses to line 19, where the best allocation is performed, by putting the piece into the corner in its chosen orientation. The chosen piece is then removed from the list of pieces that need to be packed, and the corner structure is updated because the new piece will create new corners.

The final section of code contains two conditional statements, on lines 22 and 25. The first creates a new bin if the heuristic chose to put a piece into the empty bin. When solving a bin packing problem, the framework always keeps an empty bin available to the heuristic. When solving a knapsack problem, only one bin is available, and this will eventually be filled up (with pieces and filler pieces) so that there are no corners left. The second conditional statement checks for this condition and breaks from the main loop when such a solution is found.

In summary, when the algorithm exits from the main while loop, the packing is complete and the heuristic has been used to form a solution to either a bin packing or knapsack problem. It has done this by choosing which piece to pack next, and into which corner, by returning a value for each possible combination. The combination with the highest value is taken as the heuristic's choice.

TABLE 7.1: The functions and terminals used in chapter 7, and descriptions of the values they return

	Name	Description
Functions	+	Add
	-	Subtract
	*	Multiply
	%	Protected divide
Terminals	Volume	The volume of the piece
	Value	The value (or profit) of the piece
	XYWaste	(The X-dim of the current corner's XY surface minus the piece's X-dim) PLUS (the Y-dim of the current corner's XY surface minus the piece's Y-dim)
	XZWaste	(The X-dim of the current corner's XZ surface minus the piece's X-dim) PLUS (the Z-dim of the current corner's XZ surface minus the piece's Z-dim)
	YZWaste	(The Y-dim of the current corner's YZ surface minus the piece's Y-dim) PLUS (the Z-dim of the current corner's YZ surface minus the piece's Z-dim)
	CornerX	The X coordinate of the current corner
	CornerY	The Y coordinate of the current corner
	CornerZ	The Z coordinate of the current corner

7.4.2 The Structure of the Heuristics

A heuristic is a tree structure, containing function and terminal nodes. At each decision point, the heuristic returns a value for each of the valid allocations that could be performed. As each allocation represents a combination of a piece, orientation, and corner, the terminal nodes describe these three elements of the allocation and allow the heuristic to distinguish between different allocations.

Table 7.1 summarises the twelve functions and terminals and the values they return, the functions are shown in the upper four rows, and the terminals in the lower eight rows. The functions are the four standard arithmetic operators. The protected divide function changes a zero denominator to 0.001, as explained in section 6.3.2 of chapter 6.

The 'volume' terminal returns the product of the current piece's three dimensions. This allows the heuristic to differentiate between pieces in the same corner. The alternative to this would be to create a separate terminal for each dimension of the piece. While this

would provide more detail, it would take more effort during the evolution to arrange them into useful configurations. For this reason, there is just one terminal representing overall size, as this would be likely to be the most common use of the three such terminals. The 'value' terminal returns the value of the piece, which is a key property of each piece in the knapsack problem.

The three 'waste' terminals give the heuristic information on how well the piece fits into the corner under consideration. There is one for each surface that meets to form the corner, XY, XZ and YZ. They are calculated by summing the difference between the length of the surface and the length of the piece, in both directions of the surface. So it returns zero if the piece fits onto the surface exactly, and it gets higher when the two dimensions of the piece are smaller than the two dimensions of the surface.

These terminals are potentially useful in a number of situations. For example, in one instance it may be beneficial to make the pieces line up in one plane, but it might not be important to match up the pieces in the other two planes. In another instance all the pieces may be roughly the same dimensions, with slight variations, and it may be important to orient them all the same way. The waste terminals could facilitate such processes because they give the heuristic access to information on the fit of the piece to the pieces that intersect to form the corner, in all three planes.

Finally, there are three terminals, cornerX, cornerY, and cornerZ, which provide the heuristic with information about the position of the corner in the container. They return the relevant coordinate of the corner. For example, cornerY returns the Y coordinate of the current corner.

7.4.3 How the Heuristics are Evolved

Section 7.4.1 explained how one heuristic is applied to a problem instance to obtain a solution. This section explains how the heuristics themselves are evolved by the genetic programming algorithm, and consists of two sub-sections. The genetic programming parameters are described in the first part, and the fitness function is explained in the second part. Compared to previous chapters, many more data sets are employed to evolve the heuristics in this chapter. For this reason they are described separately in section 7.5.

TABLE 7.2: Initialisation parameters of the experiments of chapter 7

Population size	1000
Generations	50
Crossover probability	0.85
Mutation probability	0.1
Reproduction probability	0.05
Tree initialisation method	Ramped half-and-half
Selection method	Tournament selection, size 7

Parameters

Table 7.2 shows the parameters used in the genetic programming runs. The mutation operator is point mutation, using the 'grow' method explained in [164], with a minimum and maximum depth of five. The crossover operator produces two new individuals with a maximum depth of 17. These are standard default parameters provided in the genetic programming implementation of the ECJ (Evolutionary Computation in Java) package employed for the genetic programming implementation for the experiments of this chapter.

Fitness Functions

A fitness is assigned to each heuristic in the population, based on the quality of the packing it produces. The quality of the packing is necessarily calculated differently for bin packing and knapsack instances.

Bin Packing Fitness

For bin packing problems, we use the fitness function shown in equation 7.3, taken from [97]. In equation 7.3, n = number of bins, $fullness_i$ = sum of the volumes the pieces in bin i, and C = bin capacity (total volume). Lower fitness values are better.

$$Fitness = 1 - \left(\frac{\sum_{i=1}^{n} (fullness_i/C)^2}{n} \right) \tag{7.3}$$

This fitness function puts a premium on bins that are nearly or completely filled. Importantly, the fitness function avoids the problem of plateaus in the search space, which occur when the fitness function is simply the number of bins used by the heuristic. We

subtract from one as the term in brackets equates to values between zero and one and we are interested in minimising the fitness value.

Knapsack Fitness

For knapsack problems, the fitness of a heuristic is the total value of all of the pieces that have been chosen by the heuristic to be packed into the single knapsack. The reciprocal of this figure is then taken as the fitness, because the genetic programming implementation treats lower fitness values as better. This fitness function is shown in equation 7.4, where n represents the number of pieces in the instance, x_j is a binary variable indicating if the piece j is packed in the knapsack, and v_j represents the value of the piece j.

$$Fitness = \frac{1}{\sum_{j=1}^{n} v_j x_j} \tag{7.4}$$

7.5 The Data Sets

We have tested our methodology on a comprehensive selection of 18 data sets from the literature. We list them in this section, and in table 7.3. It is common, in the literature, for the same cutting and packing instances to be used to test both bin packing and knapsack methodologies. Instances that are originally created for the knapsack problem can be used as bin packing instances by ignoring the value of each piece, and packing all the pieces in the instance into the least bins possible. Instances originally intended as bin packing problems do not specify a value for each piece because this is irrelevant, so to use them as knapsack instances it is usual for practitioners to calculate the volume of each piece, and to use that as its value.

The instances have been used in a large number of papers. In table 7.3 we give the references where the instances have been used to obtain the results we have compared against in this thesis. They use the same set of constraints, meaning we can compare fairly with these results. The remainder of this section gives a more detailed account of where and how the data sets have been employed.

TABLE 7.3: A summary of the 18 data sets used in the experiments of this chapter

	Instance Name	Number of Instances	Used For		References
			Bin Packing	Knapsack	
1D	Uniform	80	√	×	[96]
	Hard	10	√	×	[246, 248]
2D	beng	10	√	×	[21, 201]
	Okp	5	×	√	[221]
	Wang	1	×	√	
	Ep30	20	×	√	
	Ep50	20	×	√	
	Ep100	20	×	√	
	Ep200	20	×	√	
	Ngcut	12	√	√	[221, 201]
	Gcut	13	√	√	
	Cgcut	3	√	√	
3D	Ep3d20	20	×	√	[221]
	Ep3d40	20	×	√	
	Ep3d60	20	×	√	
	Thpack8	15	×	√	[211, 25, 66, 31]
	Thpack9	47	√	√	[144, 150, 25, 92, 185]
	BandR	700	×	√	[221, 31, 32, 184, 185]

7.5.1 One Dimensional Instances

Uniform

This is a data set consisting of 80 instances with piece sizes distributed uniformly between 20 and 100 inclusive. The set is split into four subsets, with 120, 250, 500 and 1000 pieces respectively. The bin capacity is 150 for every instance. These instances are used in [96].

Hard

This is a data set consisting of 10 instances, each with 200 pieces distributed between 20,000 and 35,000, with a bin capacity of 100,000. These instances were first proposed in [246], where three instances were solved to optimality. The seven remaining instances were solved to optimality in [248].

7.5.2 Two Dimensional Instances

Bengtsson

Originally defined in [21], the ten instances in this data set were also used in [201]. All ten instances were solved to optimality.

Okp

This data set is defined in [102]. It is also used in [103] and [57], and solved to optimality in all cases, where the orientation of the pieces is fixed. The results of [221] are used here for comparison because they are the only set of results in the literature where 90° piece rotations are allowed, and optimal results have not been obtained.

Wang

This data set is defined in [275], and solved using only guillotine cuts. We compare to the results in [221] because they allow 90° piece rotations.

*Ep**

This collection of data sets is defined in [221]. There are four data sets of 20 instances, with 30, 50, 100, and 200 pieces to pack respectively. In each of these sets there are five groups of four instances, with the pieces in each group having different characteristics, tall, wide, uniform, square and diverse. In each group of four there are two that are 'clustered', with twenty different pieces that are repeated, and two that are 'random', with independently generated pieces. Of the two instances of each, the bin area is either 25% or 75% of the total area of the pieces.

Ngcut

This data set was originally defined in [17] as a knapsack problem, and all instances were solved to optimality for the case where no rotations of pieces are allowed. All twelve instances have also been solved to optimality for the bin packing problem in [201], where the piece orientation is fixed, and an exact algorithm is used. These are the only bin packing results in the literature with this data set, and as such we limit the pieces to one orientation

for this instance to keep the comparisons fair. We also compare to the knapsack results obtained in [221] on this data set.

Gcut

This data set was originally defined in [16] for the two cutting problem with guillotine cuts. Hadjiconstantinou and Christofides [128] used the data set for constrained non-guillotine cutting, fixing the orientation of pieces. The set has also been employed in [102] and [103] with fixed orientation of pieces. We compare to the optimal bin packing results of [201], fixing the orientations of the pieces as in that paper. We also compare to the heuristic knapsack results of [221].

Cgcut

This data set was defined as a knapsack problem in [63], where the problem is constrained to guillotine cuts. The data set has since been used in [139, 103] with fixed orientation of pieces. We compare to the bin packing results in [201], and limit the orientation of pieces as is done in that paper. We also compare to the knapsack results obtained with a heuristic method in [221], where the pieces are free to be rotated by 90°.

7.5.3 Three Dimensional Instances

*Ep3d**

The Ep3d collection of data sets is defined in [221], and is similarly constructed to the Ep collection described in section 7.5.2. There are three data sets of 20 instances, with 20, 40 and 60 pieces to pack. Within each data set there are 5 instance groups, each group has either flat, long, cube, uniform, or diverse pieces. In each of these groups of four instances, two are 'clustered' and two are 'random' as described for the Ep set. One of these two has its container volume at 50% of the total volume of pieces, and the other one has its container volume at 90%.

Thpack8

This data set was originally defined in [191], and later used in [211, 25, 66]. Bortfeldt and Gehring have tested both their CBUSE and CBGAS methodologies on the data set also, and the results are reported in [31].

Thpack9

Thpack9 was defined originally as a data set of three dimensional bin packing problems in [150]. We compare our results to [150], and later work in [25, 92]. The work of Lim and Zhang [185] is also compared to in this paper. As a knapsack problem, the first 44 instances in the set have been used in [184]. Huang and He [144] achieve better results and use all 47 instances.

BandR

This is the largest data set, with 700 instances. In all instances, the container size is the same, length 587, width 233, and height 220 (the dimensions in cm of a standard 20ft ISO container). The data set is split into seven sets of 100 instances, and the seven sets have 3, 5, 8, 10, 12, 15, and 20 different piece types. The random generator used to create the set was defined in [25].

The data set was used in that paper to evaluate a methodology that aims to provide very stable packings, and also packings useful in multi-drop situations, where there are multiple stops where only part of the packing solution is to be unloaded. As these objectives are not considered here, there is little value in a comparison with that work, but we do compare with the later work on this data set, in [31, 184, 32, 185]. This data set is the only set which constrains the orientations of the pieces, and those constraints are also adhered to in our work, to ensure a fair comparison.

7.6 Results

We obtain results on a comprehensive suite of 18 unique data sets, the details of which are shown in section 7.5. Each data set contains a number of instances, for which numerous results exist in the literature. In this section, the results obtained by the hyper-heuristic methodology are compared to the best result in the literature for that instance. Tables 7.5 and 7.6 show the 'Ratio' for each data set, which is a value obtained by comparing our results on a data set to the best results in the literature. To get to this Ratio figure, there are three steps, described below and shown in equations 7.5, 7.6 and 7.7, and further clarified using an example of one data set in section 7.6.1.

$$InstanceAverage = \frac{\sum_{i=1}^{10} r_i}{10} \tag{7.5}$$

$$InstanceRatio = \frac{InstanceAverage}{z} \tag{7.6}$$

$$Ratio = \frac{\sum_{i=1}^{m} InstanceRatio_i}{m} \tag{7.7}$$

For every instance, we perform ten runs, each with a different random seed. This generates ten different heuristics. We calculate the average performance of the ten heuristics for the instance, and name this value the 'InstanceAverage'. This is shown in equation 7.5, where r_i is the result of run i.

For each instance, we then calculate the ratio of the InstanceAverage over the best result in the literature, and name this value the 'InstanceRatio'. This is shown in equation 7.6, where z is the best result in the literature for the given instance.

Each instance in a set has an InstanceRatio. The 'Ratio' is the average InstanceRatio of all the instances in the set. This is shown in equation 7.7, where i is the instance number, and m is the number of instances in the data set. Tables 7.5 and 7.6 show the Ratio for each data set. The standard deviation reported is for the distribution of InstanceRatio values, of which there will be one for every instance in the set.

As the bin packing problem is a minimisation problem, a Ratio lower than 1 means our result is better than the best result in the literature. Conversely, the knapsack problem is a maximisation problem, so a Ratio higher than 1 means our result is better. To keep the results tables consistent, we convert the Ratios to percentage values, which represent our improvement over the best results in the literature. For example, a Ratio of 1.023 as a bin packing result represents a -2.3% improvement, because it means that, in the results we obtained, we use 2.3% more bins on average. This percentage figure is shown in tables 7.5 and 7.6. A positive percentage means the results are better than the results from the literature, and a negative percentage means the results are worse. The results of this table will be discussed in sections 7.6.2 and 7.6.3, after an example calculation of the Ratio value is performed in section 7.6.1.

7.6.1 An Example Calculation

In this section, we will show the individual results for each instance in the Thpack9 bin packing data set, to further clarify how the Ratio is obtained for a set.

Table 7.4 shows our results in detail. There are six algorithms from four different papers that use this data set, and their results are shown in the columns A-F in the table.

TABLE 7.4: The individual instance results for the Thpack9 bin packing data set

Instance	A	B	C	D	E	F	Best in Literature	GPHH Best	GPHH Average	Instance Ratio
thpack9-1	26	27	27	27	26	**25**	25	25	25	1.000
thpack9-2	11	11	11	11	**10**	**10**	10	10	10	1.000
thpack9-3	20	21	26	21	22	**19**	19	19	19.5	1.026
thpack9-4	27	29	27	29	30	**26**	26	26	26	1.000
thpack9-5	65	61	59	55	**51**	**51**	51	51	51	1.000
thpack9-6	**10**	**10**	**10**	**10**	**10**	**10**	10	10	10	1.000
thpack9-7	**16**	**16**	**16**	**16**	**16**	**16**	16	16	16	1.000
thpack9-8	5	**4**	**4**	**4**	**4**	**4**	4	4	4	1.000
thpack9-9	**19**	**19**	**19**	**19**	**19**	**19**	19	19	19	1.000
thpack9-10	**55**	**55**	**55**	**55**	**55**	**55**	55	55	55	1.000
thpack9-11	18	19	25	17	18	**16**	16	16	16.2	1.013
thpack9-12	55	55	55	**53**	**53**	**53**	53	53	53	1.000
thpack9-13	27	**25**	27	**25**	**25**	**25**	25	25	25	1.000
thpack9-14	28	**27**	28	**27**	**27**	**27**	27	27	27	1.000
thpack9-15	**11**	**11**	15	12	12	**11**	11	11	11	1.000
thpack9-16	34	28	29	28	**26**	**26**	26	26	26	1.000
thpack9-17	8	8	10	8	**7**	**7**	7	7	7.1	1.014
thpack9-18	3	3	**2**	**2**	**2**	**2**	2	2	2	1.000
thpack9-19	**3**	**3**	**3**	**3**	**3**	**3**	3	3	3	1.000
thpack9-20	**5**	**5**	**5**	**5**	**5**	**5**	5	5	5	1.000
thpack9-21	24	24	26	24	26	**20**	20	20	20	1.000
thpack9-22	10	11	11	**9**	**9**	**9**	9	9	9	1.000
thpack9-23	21	22	22	21	21	**20**	20	20	20	1.000
thpack9-24	6	6	7	6	6	**5**	5	5	5.6	1.120
thpack9-25	6	**5**	**5**	6	**5**	**5**	5	5	5	1.000
thpack9-26	**3**	**3**	4	**3**	**3**	**3**	3	3	3	1.000
thpack9-27	**5**	**5**	**5**	**5**	**5**	**5**	5	5	5	1.000
thpack9-28	10	11	12	11	10	**9**	9	10	10	1.111
thpack9-29	18	**17**	23	18	18	**17**	17	17	17.7	1.041
thpack9-30	24	24	26	**22**	23	**22**	22	22	22.5	1.023
thpack9-31	13	13	14	13	14	**12**	12	13	13.4	1.117
thpack9-32	5	**4**	**4**	**4**	**4**	**4**	4	4	4	1.000
thpack9-33	5	5	5	5	5	**4**	4	5	5	1.250
thpack9-34	9	9	**8**	**8**	9	**8**	8	8	8	1.000
thpack9-35	3	3	3	**2**	**2**	**2**	2	3	3	1.500
thpack9-36	18	19	**14**	18	**14**	**14**	14	14	14	1.000
thpack9-37	26	27	**23**	26	**23**	**23**	23	23	23	1.000
thpack9-38	50	56	**45**	46	**45**	**45**	45	45	45	1.000
thpack9-39	16	16	18	**15**	**15**	**15**	15	15	15	1.000
thpack9-40	**9**	10	11	**9**	**9**	**9**	9	8	8	0.889
thpack9-41	16	16	17	16	**15**	**15**	15	15	15	1.000
thpack9-42	**4**	5	5	**4**	**4**	**4**	4	4	4	1.000
thpack9-43	**3**	**3**	**3**	**3**	**3**	**3**	3	3	3	1.000
thpack9-44	4	4	4	4	4	**3**	3	3	3	1.000
thpack9-45	**3**	**3**	**3**	**3**	**3**	**3**	3	3	3	1.000
thpack9-46	**2**	**2**	**2**	**2**	**2**	**2**	2	2	2	1.000
thpack9-47	4	**3**	4	**3**	**3**	**3**	3	3	3	1.000
										Ratio = 1.023482

TABLE 7.5: Summary of bin packing results, and the percentage improvement over the best results in the literature

	Instance Name	Ratio	S.D.	Percent Improvement
1D	Uniform	1.000	0.003	0
	Hard	1.004	0.007	−0.4
2D	Beng	1.000	0.053	0
	Ngcut	1.000	0.000	0
	Gcut	1.012	0.041	−1.2
	Cgcut	1.000	0.000	0
3D	Thpack9	1.023	0.086	−2.3

Column A shows the results of [150]. Columns B and C show the results of the two algorithms presented in [25]. Columns D and E show the results of the two algorithms 'E-seq' and 'E-sim' presented in [92], and column F shows the results of [185].

The best results are highlighted in bold and summarised in the 'Best in Literature' column. 'GPHH Best' and 'GPHH Average' show the best and average results respectively from our ten runs on each instance. The InstanceRatio column is the result of equation 7.6 for each instance, in other words the result of 'GPHH Average' divided by the 'Best in literature'. The Ratio for the data set is given at the foot of the table under the InstanceRatio column. This table shows how we arrive at the Ratio for the data set, it is the average of all of the InstanceRatio values of the data set.

7.6.2 Bin Packing Results

The discussion in this section refers to the results reported in table 7.5.

One Dimensional

For the 'Uniform' data set, we compare to the results of Falkenauer [96]. The results of the evolved heuristics are one bin worse than Falkenauer's results for three instances out of 80. The heuristics obtain results one bin better for two instances out of the 80, achieving the proven optimum results in those cases. Also, for each instance, our average of the ten runs is never more than one bin worse than our best result, so in this respect the results are consistent.

For the 'Hard' data set, we compare to Schwerin and Wäscher [248] who have solved the instances to optimality. The evolved heuristics achieve the optimal result for all

TABLE 7.6: Summary of knapsack results, and the percentage improvement over the best results in the literature

	Instance Name	Ratio	S.D.	Percent Improvement
2D	Okp	1.012	0.019	+1.2
	Wang	1.000	0.000	0
	Ep30	0.991	0.018	−0.9
	Ep50	1.004	0.014	+0.4
	Ep100	0.974	0.072	−2.6
	Ep200	1.024	0.015	+2.4
	Ngcut	0.957	0.143	−4.3
	Gcut	1.012	0.062	+1.2
	Cgcut	0.983	0.016	−1.7
3D	Ep3d20	1.130	0.107	+13.0
	Ep3d40	1.102	0.100	+10.2
	Ep3d60	1.060	0.067	+6.0
	Thpack8	0.995	0.014	−0.5
	Thpack9	1.007	0.000	+0.7
	BandR	0.971	0.004	−2.9

but one instance of the set of ten, where our best result uses one bin more than the optimal. For seven of the instances, the evolved heuristics find the optimal result in all ten runs.

Two Dimensional

For the 'Beng' data set, the results are compared to the exact methods in [21, 201]. For the 'Ngcut', 'Gcut', and 'Cgcut' sets, we compare the results to [201]. The results that have been reported for these four data sets are obtained without allowing rotations of the pieces, so for a fair comparison we applied the same constraint.

Martello and Vigo [201] do not find a result for the eighth gcut instance. The heuristic evolved for this instance does find a result, but it is not included in the calculation of the Ratio for this data set because an InstanceRatio cannot be calculated without a result from the literature.

Three Dimensional

Thpack9 is the only three dimensional bin packing data set, and the results can be seen in detail for this set in table 7.4. For 42 instances out of the 47, our best heuristic evolved from the ten runs matches the best result from the literature. There are four instances in

which we use one more bin than the best result, and one instance in which we beat the best result by one bin. The average bins used in each of the ten runs is never more than 0.7 greater than the best result of the ten runs, so the heuristics evolved are of consistently good quality for each instance.

7.6.3 Knapsack Results

The discussion in this section refers to the results reported in table 7.6.

Two Dimensional

We compare all our two dimensional knapsack results to the results in [221], where the four new 'Ep*' instances are defined to test their heuristic. They also apply their heuristic approach to five older data sets, previously solved to optimality with exact methods, but still useful to compare heuristic methods which are not guaranteed to achieve the optimal result.

The results on the nine problem sets show that there is no real difference between the performance of the evolved heuristics and the performance of the heuristic method used in [221]. There are five sets with a Ratio just greater than one, and four just with less than one. In the ngcut set, the high standard deviation is because of the 6th instance of the set where the ratio is 0.5. Without this outlier the standard deviation of ngcut would be less than 0.01.

Three Dimensional

There are three subsets in the ep3d set. In the first set, with instances of 20 pieces, the results of the evolved heuristics are better in every instance than the results in [221]. In the second set, the results are better in all but two instances, and in the third set all but four of the evolved heuristics obtain better results.

Thpack8 has 15 instances. In 13 of those, we match the best result for the instance from five papers that report results. However, in the 2nd and 6th instances in the set, the evolved heuristics could not reach the results of the CBUSE method of [31]. Note though that the result for the 2nd instance is second only to CBUSE, and the result for the 6th instance is third best.

Over the 47 instances of Thpack9, we obtain an average space utilisation of 95.2%, which is compared to the 94.6% of [144]. We achieve 100% space utilisation in 14 instances out of the 47. The individual results for the 47 instances of this set are not reported in the literature, so in this data set, equations 7.5, 7.6 and 7.7 could not be calculated. Therefore, in the Ratio column of table 7.6 for this data set, we report the ratio of our average space utilisation over the average space utilisation of [144]. This also means that the standard deviation is not reported because InstanceRatio values could not be calculated for each instance of this set.

There are 5 papers that report results for the BandR data set. As explained in section 7.5.3, there are 700 instances, split into 7 subsets of 100. Therefore, unsurprisingly, the results for the individual instances have not been reported in previous papers. Only the averages of the 7 subsets have been reported, along with an overall average for the 700 instances. So in this data set, similar to Thpack9, we could not compare to the best results from the literature for each individual instance.

So for each of the seven subsets, we compile the best average space utilisation reported in the literature, from the five papers that report results. We calculate the equivalent of an 'InstanceRatio' for each subset, by dividing our average for the subset by the best result for the subset. Then the Ratio is calculated with equation 7.7 with m set to seven, rather than 700 as would be expected. So for the BandR data set, we are comparing against the best technique on each of its seven subsets, rather than on each instance.

7.7 Conclusion

In this chapter, we have described a genetic programming methodology that automatically generates a heuristic for any one, two, or three dimensional knapsack or bin packing problem. This is the first hyper-heuristic system to be applied to three dimensional packing. The goals of this chapter are a natural progression from the previous four chapters, as the problem domains have increased in dimensionality from one to three. The results of the automatically designed heuristics can at least equal the results obtained by complex human designed heuristics, in the packing problems addressed here. This is especially significant as these packing problems are well studied, and the human designed heuristics obtain very close to optimal solutions for some of the data sets.

The good results are possible because the experiments evolve one heuristic for

one instance. The heuristic can therefore specialise on that instance. The heuristics are 'disposable' in this sense, because they are evolved for the sole purpose of obtaining a solution to one instance, and are not meant to be applied to any another instances.

This highlights two areas of further work. Firstly, it will be necessary to investigate the circumstances where the heuristics can be reused. The data-sets used here have varying piece sizes, so maybe if the pieces are more uniform then the heuristics would be able to generalise better. Also, the heuristics will have to be evolved on more training instances. The re-usable heuristics for one dimensional bin packing evolved in chapters 4 and 5 were evolved on a training set of 20 instances. It may be that less training instances could be used, as the heuristics take longer to pack the three dimensional instances, and 20 may be too many. The terminals may have to be changed to some which provide more general information. Particularly, the three waste terminals could be changed to some which better describe the relationship of the current piece with its potential surroundings.

Secondly, further work could ascertain whether a heuristic evolved on three dimensions can be successful on one and two dimensional instances. This would mean one single heuristic would have been designed by evolution to solve 1D, 2D and 3D packing instances. This is a task that no human created constructive has been designed to perform currently.

Automatic heuristic generation is a relatively new area of research. The traditional method of solving packing problems is to obtain or generate a set of benchmark instances, and design a heuristic that obtains good results for that data set. This process of heuristic design can take a long time, after which the resulting heuristic is specialised to the benchmark data set. Recently, metaheuristics have been developed which operate well over a variety of instances. Designing a metaheuristic system, and optimising its parameters, is a process that can take many more hours of research. A system has not been presented before that can operate over these different problem domains, and maintain a human-competitive quality of results. For example, before the work presented here, a metaheuristic system developed solely for three dimensional packing instances could not operate on one dimensional instances.

Therefore, in addition to showing that human competitive heuristics can be automatically designed, a further contribution of this chapter is to present a system that can generate a solution for any one, two, or three dimensional knapsack or bin packing problem instance, without the need to tune parameters for different dimensions/instances. All that is required is to provide the problem instance file, and specify whether it is a bin packing or

knapsack instance. This system does take more computational effort to generate a solution to an instance, but this is the price paid for generality over three packing domains. One of the goals of hyper-heuristic research is to "raise the level of generality" of optimisation methods. We conclude that the methodology presented here represents a more general system than those currently available, due to its performance on problems in the one, two and three dimensional cases of two packing problem domains.

Chapter 8

Conclusion

8.1 Context

A hyper-heuristic is a heuristic that searches a space of heuristics, rather than a solution space directly. The goal of such research is to develop systems which are more general than those currently available [47]. The first hyper-heuristic system was developed in the early 1960s [106, 107]. Since then, the area has received little attention up until recently, where there has been an increase in hyper-heuristic research activity. The need is now clear for more general systems. Most hyper-heuristic research to date has involved intelligently selecting a heuristic from a pre-defined set. The alternative, which has received less attention, is to enable a hyper-heuristic to design a heuristic from a set of potential components. This process of automatic heuristic generation is the subject of this thesis, and it is of particular interest as suitable heuristics may not exist for a problem, and the cost of a bespoke heuristic system is often beyond what a small organisation can afford. An automated heuristic generation system has the potential to spread the high cost over a number of customers.

This thesis has presented and analysed a hyper-heuristic methodology which automates the heuristic design process for one, two and three dimensional bin packing and knapsack problems. The heuristics designed by this methodology produce results competitive with, and often superior to, those obtained by human created heuristics. Indeed, the genetic programming system of chapter 7 represents the first packing algorithm in the literature able to claim human competitive results in such a wide variety of packing domains, without requiring *any* change in parameters. The same cannot be claimed of other packing systems currently in the literature, which are developed to operate on single problem domains. The system presented in this thesis searches a space of heuristics which can be built

from a set of components. The components that are relevant to the problem at hand can be chosen, and others can be discarded, which means that the system can be more general and maintain a high quality of results.

In the field of packing problems, there is a large literature on human created heuristics. In contrast, no work has been carried out on *automating* the heuristic design process for packing. However, automatic heuristic generation has been studied relatively recently by Fukunaga for the SAT problem [112, 113, 114]. Fukunaga's approach stems from the fact that many existing SAT heuristics are composites, in the sense that they can be seen as different arrangements of the same set of components. These components are identified and employed as the functions and terminals of a genetic programming system named CLASS, which evolves a heuristic that decides which variable to flip next. Bader-El-Din and Poli [9] observe that this results in heuristics which are relatively slow to execute, because they are composites of heuristics in early generations. Bader-El-Din and Poli present a different heuristic generation methodology for SAT, which makes use of traditional crossover and mutation operators to produce heuristics which are more parsimonious, and faster to execute. A grammar is defined, which can express four existing human created heuristics, and allows significant flexibility to create completely new heuristics.

Another system for automatically designing heuristics is presented by Keller and Poli [159], for the travelling salesman problem. They use a grammar based linear genetic programming technique to evolve parsimonious metaheuristics. While the genetic programming technique they employ is different, they evolve a heuristic for each instance, similar to the methodology used in chapters 6 and 7.

Work presented by Geiger et al. in [118] is highly relevant to the work presented here. They use genetic programming to evolve priority dispatching rules for a single machine shop environment, and more complex shop configurations. The format of these heuristics is very similar to those presented in this thesis, in the sense that they are tree structures with terminals which represent different attributes of the job and machine. The dispatching rules learn relationships between the job and machine attributes, in a similar way to how the heuristics presented here learn beneficial relationships between piece and bin attributes.

There are many more examples of the automatic generation of heuristics in the literature, and the area is receiving progressively more attention. For example, in production scheduling [151, 140, 260, 84], cutting and packing [172], the travelling salesman problem [213], and function optimisation [212, 214, 259]. The activity in this area reflects the need

for more general systems, which can effectively automate as much of the heuristic design process as possible.

8.2 Summary of Work

8.2.1 Chapters 3 - 5

Chapters 3 to 5 developed and improved the hyper-heuristic methodology over a series of one dimensional packing problems. Chapter 3 introduced the fundamental methodology that would be used throughout the thesis, that of a heuristic returning a score for a number of potential locations for a piece. In chapter 3, the piece was placed into the location which received the first score greater than zero. This chapter showed that the quality of heuristics designed in this way could be as good, or better than, the human designed first-fit heuristic.

Chapter 4 refined the methodology further by allowing all possible locations to be considered by the heuristic. The piece was put into the location which received the highest score overall, rather than being placed into the first location which achieved a certain threshold. This greater flexibility allowed the evolved heuristics to beat the human designed best-fit heuristic. This chapter also showed that the heuristics can be reused on unseen problem instances. They achieve the best results on new problem instances of the same class as those they were evolved on. Also, it was shown that the heuristics become specialised on the class that they were evolved on. A hierarchy of performance is formed, where heuristics are specialists on a particular sub-problem, or general enough to work on all sub-problems. It was shown empirically that there is a trade off between performance and generalisation. The third conclusion of chapter 4 was that a heuristic could produce unexpected results if it was applied to instances with piece sizes different to those which it had seen during its evolution. It is not only the functions and terminals of an evolved heuristic which produce its behaviour, it is also their relationship with the sizes of the pieces. Therefore, the structure of an evolved heuristic only produces its behaviour on instances of the same class as those it has seen before.

Chapter 5 investigated the scalability of the evolved heuristics. The experiments generated heuristics for small instances, and then applied them to much larger unseen instances. It was found that the heuristics maintain their performance beyond the size of their training set instances. The long term performance of the heuristics is increased when the training set is larger. It was explained that, at the start of the packing, the

heuristics perform worse than best-fit, but the heuristics are shown to be performing better than best-fit as the packing progresses beyond half the number of pieces in the training set. It appears as if the heuristics take a long term view, and perform sub-optimally at the beginning, in order to be able to outperform best-fit by the time all of the pieces have been packed. However, the heuristics have no information regarding the current stage of the packing, and therefore cannot change their behaviour to begin packing more tightly at the final stage. The results were explained by the logic that, if more bins are kept open, then a wider distribution of open bins is available, and less space is wasted by closing a bin too soon.

8.2.2 Chapter 6

The two dimensional strip packing problem was investigated in chapter 6. The representation on which the heuristics operate took inspiration from the best-fit constructive heuristic, presented in [38]. As the best-fit heuristic can pack any piece next from all of those which are still available, chapter 6 introduced a new element to the framework in which the heuristics operate, to give the heuristics the same freedom. It was shown that the piece order could be rendered virtually irrelevant when using this hyper-heuristic methodology. This changes the problem to an offline problem, rather than online, because all of the pieces are made known before the packing begins. This is similar to a production situation where all of the required shapes are known before the cutting pattern for the stock sheets is decided. The results of chapter 6 show that it is possible to evolve heuristics that obtain results which are better than the results of human designed heuristics. One conclusion to be drawn from this work is that, because of the nature of how the heuristics are applied to pack an instance, the process of evolving a heuristic is time consuming. A number of potential areas for future research are specified in section 8.3, which will address this important issue.

8.2.3 Chapter 7

Chapter 7 employed the hyper-heuristic methodology to automatically design heuristics for the three dimensional bin packing and knapsack problems. In a similar way to that presented in chapter 6, the heuristics gave a score to each potential location for each piece, and performed the allocation which received the highest score. The available locations in the three dimensional representation were corners, where three perpendicular surfaces

met. A contribution of this chapter was that the hyper-heuristic methodology was capable of evolving heuristics for one and two dimensional packing instances, in addition to three dimensional instances. The system also evolved heuristics for both the bin packing and knapsack problems in all three dimensions. Importantly, no parameter changes were required between changes from one problem domain to another. The results were shown to be competitive with the results of human created heuristics. This is a significant result considering that no other system in the literature can obtain results over such a wide range of problem domains.

We have already stated that the need is now clear for more general systems, and we believe that chapter 7 raises important issues concerning what we refer to by the 'generality' of a system. In chapter 7, the system was capable of finding a solution to more problem instances than any other packing system in the literature. Clearly this shows the generality of the genetic programming approach presented in this thesis. However, an important goal of current hyper-heuristic research is to automate the design process for heuristics that can also be reused after they have been evolved. Humans have proven that they are adept at analysing a group of problem instances and then designing heuristics which exploit the structure of those problems, but this is currently a costly process. We believe that this approach has the potential to reduce the cost of designing bespoke heuristics, through further automation of the design process. These automatically designed heuristics may not be general, they may be specialised to a given type of packing problem, but the hyper-heuristic system that designed the heuristics can be referred to as general if it can produce heuristics for a wide range of different problem types.

Note that the heuristic design process is, of course, not *completely* automated with such a system. This would represent a 'general solver', which can not exist currently. However, this hyper-heuristic approach does represent a change of emphasis regarding the role of the human heuristic designer. With a hyper-heuristic system which automates part of the design process, the human designer need only identify good building blocks for heuristics, and the time consuming task of combining them in an intelligent way is passed to the computer.

8.3 Extensions and Future Work

We suggest here some research directions in order to extend the work presented in this thesis.

8.3.1 Evolving Reusable Heuristics for Two and Three Dimensional Packing

Chapters 3 to 5 investigated the reuse of the evolved heuristics for one dimensional bin packing. The same could be investigated of the two and three dimensional heuristics. While these heuristics obtain very good results, they take a long time to evolve due to the many heuristics that must be evaluated in each generation. Being able to reuse the heuristics after they have been evolved would further justify the relatively large run times. As stated in the conclusion of chapter 7, the data-sets used here have varying piece sizes, so it may be necessary to generate more uniform instances for the heuristics to generalise successfully. Also, the heuristics will have to be evolved on more training instances in order for them to generalise. The re-usable heuristics for one dimensional bin packing, evolved in chapters 4 and 5, were evolved on a training set of 20 instances. It may be that less training instances could be used, as the heuristics take longer to pack the three dimensional instances, and 20 may be too many. Another related question for research is whether a validation set of instances is necessary when evolving heuristics for packing, as is the standard method for preventing overfitting to the training set.

It is left as a broad area for future work to determine whether the existing functions and terminals are sufficient to evolve generalisable heuristics, or if they would need to be modified to incorporate more general information. For example, the 'piece width' terminal currently encodes the absolute value of the width of the piece, but it may be necessary to redefine this terminal. The redefined terminal may encode the piece width as a fraction of the sheet width, or as a fraction of the maximum piece width in the instance. Then the heuristic may be more applicable to new problem instances. To express the issue a different way, if one was to take a problem instance, and create a new instance by reducing the size of all the dimensions by half, then one would expect that applying a heuristic to both instances would produce two solutions which look identical. Currently, because the terminals encode absolute values, it is not clear whether an evolved heuristic would produce identical results for instances which are scaled up or down.

8.3.2 Specialisation and Robustness of Two Dimensional Packing Heuristics

The work in chapter 6 also indicates another area of future work. In chapter 4, section 4.4.1, we presented an example of an evolved heuristic for one dimensional bin packing which performed poorly on new instances with pieces of a given distribution that it had not seen before. It performed poorly because it had evolved to make use of ratios (for example, of piece size to bin capacity) that did not exist in the new instances. It is our intuition that the same phenomenon would happen for two dimensional packing, and that the evolved heuristics would perform well on new instances of the same class as the training set, at the expense of their performance on other classes of instance. This area of future work could also contribute to defining the characteristics of what makes two dimensional instances similar or different, as one could label instances as conceptually 'different' if a heuristic performs well on some at the expense of the others. While this thesis has addressed many of the issues in evolving heuristics for two dimensional packing, there is still much more potential for research in this area.

8.3.3 Intelligently Select which Allocations to Consider

The conclusion of chapter 6 suggested that reducing the run times of the system would be beneficial. This could be achieved by focussing further work on evolving heuristics that intelligently select which allocations to consider, rather than evaluating *all* those available. This has the potential to reduce the time taken to evolve a heuristic, because it is the many evaluations of the heuristics in the population that account for the large run times. However, it is likely the run times will always be relatively large, compared to other work in the literature, because this methodology must design and run the heuristics, whereas other methodologies only take into account the time taken to run. It is the design stage which is the most time intensive, whether this is done by human or machine, and this is exactly why it is potentially beneficial to research ways of automating this process.

8.3.4 Improve the Three Dimensional Packing Representation

The functions and terminals used in chapter 7 were chosen because they exploit our representation of the three dimensional packing problem. This raises a possible direction for future work. The human created 'best-fit' heuristic cannot be exactly encoded by these functions and terminals and the current representation of the problem. Therefore, it can-

not be evolved by the system as presented in this thesis, and it is possible that the space of heuristics defined by the functions and terminals may not include the best heuristics for the problem. We suggest that it would be possible to obtain superior heuristics if the representation is improved to allow more piece allocations, and if the functions and terminals are changed so that they can encode the functionality of best-fit.

8.3.5 Employ Higher Level Functions and Terminals

In [114], Fukunaga compares his type of evolved heuristic with the evolved heuristics presented in chapter 3 and in [48]. He classifies our heuristics as value selection heuristics, because they decide which value to assign to each bin and piece combination. The heuristics evolved by the CLASS system are variable selection heuristics because they decide which variable to consider next. Fukunaga goes on to say that a variable selection heuristic for our problem would be one which selects the bin and piece combination to consider next. However, we use a fixed ordering which iterates through all of the combinations. This insight suggests potential future work, which would involve constructing the heuristics from higher level components than the functions and terminals used in this thesis. For example, the heuristics consist of components which provide numerical values to describe details of each potential allocation. There are often many potential allocations at each decision point, and therefore many heuristic evaluations must be performed in order to ascertain the allocation with the highest score. Multiplied over the number of individuals in the population, these evaluations result in large run times. Using higher level components would mean that every allocation need not be considered, and the chosen piece and location could be selected much more quickly. Examples of such components could be a terminal that picks a random bin, or the bin with the most free space, or the set of bins that is less than 50% full. Conditional operators could also be included to make the heuristics more flexible.

8.3.6 Evolving one General Packing Heuristic

The heuristics evolved in chapter 7 all consist of the same functions and terminals regardless of whether they are being applied to a one, two or three dimensional problem. This is a consequence of the system requiring no parameter changes between runs on any domain. Therefore, the heuristics which are evolved for three dimensional packing will obtain results if applied to one or two dimensional problems. We suggest that another area for future

work would be to investigate if the heuristics can be reused on problems of a different dimensionality than those which they were evolved on. This would potentially require a large training set, containing a representative sample of instances of each type of problem. The author knows of no current human designed heuristic for this purpose. It would represent a significant result if one computer designed heuristic could be shown to work well over all of these problems.

8.4 Final Remarks

In summary, there are three main achievements of this thesis. Firstly, to present a methodology which is capable of automatically designing heuristics for packing problems. This is the first piece of work in the literature to automatically design heuristics for this problem. Secondly, to show the heuristics are of a good enough quality to compete with human designed heuristics. Thirdly, to show that one hyper-heuristic system can be general enough to automatically design heuristics for many different packing problem domains. This thesis has addressed important issues on the quality, specialisation, robustness, and reuse of automatically designed heuristics.

The area of hyper-heuristics is still relatively new, with many aspects yet to be researched. However, it is a paradigm which appears highly appropriate for certain real world applications, as the cost of bespoke implementations of search methodologies stays beyond the means of smaller organisations, who currently solve their optimisation problems by hand. As hyper-heuristic systems receive more attention, there is a possibility that they will be able to routinely generate heuristics for such organisations, reducing the cost of computer aided optimisation through automation of the heuristic design process.

References

[1] M. Adler, P. B. Gibbons, and Y. Matias. Scheduling space-sharing for internet advertising. *Journal of Scheduling*, 5(2):103–119, 2002.

[2] A. Afshar, H. Amiri, and E. Eshtehardian. An improved linear programming model for one-dimensional cutting stock problem. In *Proceedings of the 1st International Conference on Construction In Developing Countries (ICCIDCI)*, pages 51–56, Karachi, Pakistan, August 2008.

[3] J. Allen, H. M. Davey, D. Broadhurst, J. K. Heald, J. J. Rowland, S. G. Oliver, and D. B. Kell. High-throughput classification of yeast mutants for functional genomics using metabolic footprinting. *Nature Biotechnology*, 21(6):692–696, June 2003.

[4] R. Alvarez-Valdes, F. Parreno, and J. M. Tamarit. Reactive grasp for the strip-packing problem. *Computers and Operations Research*, 35(4):1065–1083, 2008.

[5] R. Alvarez-Valdes, F. Parreno, and J. M. Tamarit. A branch and bound algorithm for the strip packing problem. *OR Spectrum*, 31(2):431–459, 2009.

[6] A. Alvim, C. Ribeiro, F. Glover, and D. Aloise. A hybrid improvement heuristic for the one-dimensional bin packing problem. *Journal of Heuristics*, 10:205–229, 2004.

[7] H. Asmuni, E. K. Burke, J. M. Garibaldi, and B. McCollum. A novel fuzzy approach to evaluate the quality of examination timetabling. In *Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT'06)*, pages 82–102, Brno, Czech Republic, August 2006.

[8] Y. Azaria and M. Sipper. GP-gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, September 2005.

[9] M. B. Bader-El-Din and R. Poli. Generating sat local-search heuristics using a gp hyper-heuristic framework. In *LNCS 4926. Proceedings of the 8th International Conference on Artifcial Evolution*, pages 37–49, October 2007.

[10] M. B. Bader-El-Din and R. Poli. Grammar-based genetic programming for timetabling. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'09)*, pages 2532–2539, Trondheim, Norway, May 2009.

[11] R. Bai, E. K. Burke, and G. Kendall. Heuristic, meta-heuristic and hyper-heuristic approaches for fresh produce inventory control and shelf space allocation. *Journal of the Operational Research Society*, 59(10):1387–1397, 2008.

[12] R. Bai, E. K. Burke, G. Kendall, and B. McCollum. Memory length in hyper-heuristics: An empirical study. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling (CISched'07)*, pages 173–178, Honolulu, Hawaii, April 2007.

[13] R. Bai and G. Kendall. An investigation of automated planograms using a simulated annealing based hyper-heuristics. In *proceedings of The 5th Metaheuristics International Conference (MIC'03)*, Kyoto, Japan, August 2003.

[14] B. S. Baker, E. G. Coffman Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855, 1980.

[15] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic programming, an introduction: on the automatic evolution of computer programs and its applications.* Morgan Kaufmann, San Francisco, 1998.

[16] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *Journal of the Operational Research Society*, 36(4):297–306, 1985.

[17] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[18] J. E. Beasley and N. P. Hoare. Placing boxes on shelves: a case study. *Journal of the Operational Research Society*, 52:605–614, 2001.

[19] G. Belov and G. Scheithauer. A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths. *European Journal of Operational Research*, 141(2):274–294, 2002.

[20] G. Belov, G. Scheithauer, and E. A. Mukhacheva. One-dimensional heuristics adapted for two-dimensional rectangular strip packing. *Journal of the Operational Research Society*, 59(6):823–832, 2008.

[21] B. E. Bengtsson. Packing rectangular pieces - a heuristic approach. *The Computer Journal*, 25(3):353–357, 1982.

[22] Y. Bernstein, X. Li, V. Ciesielski, and A. Song. Multiobjective parsimony enforcement for superior generalisation performance. In *Proceedings of the Congress for Evolutionary Computation 2004 (CEC'04)*, pages 83–89, Portland, Oregon, June 2004.

[23] B. Bilgin, E. Ozcan, and E. E. Korkmaz. An experimental study on hyper-heuristics and exam scheduling. In *LNCS 3867, Proceedings of the 6th International Conference on the Practice and Theory of Automated Timetabling (PATAT'06)*, pages 394–412, Brno, Czech Republic, August 2006.

[24] M. Biro and E. Boros. Network flows and non-guillotine cutting patterns. *European Journal of Operational Research*, 16(2):215–221, 1984.

[25] E. Bischoff and M. Ratcliff. Issues in the development of approaches to container loading. *Omega*, 23(4):377–390, 1995.

[26] E. E. Bischoff. Three-dimensional packing of items with limited load bearing strength. *European Journal of Operational Research*, 168(3):952–966, 2006.

[27] E. E. Bischoff and M. D. Marriott. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research*, 44(2):267–276, January 1990.

[28] C. C. Bojarczuk, H. S. Lopes, and A. A. Freitas. Genetic programming for knowledge discovery in chest-pain diagnosis. *IEEE Engineering in Medicine and Biology Magazine*, 19(4):38–44, July 2000.

[29] A. Bortfeldt. A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. *European Journal of Operational Research*, 172(3):814–837, 2006.

[30] A. Bortfeldt and H. Gehring. Applying tabu search to container loading problems. In *Operations Research Proceedings. Selected Papers of the International Conference on Operations Research*, pages 533–538. Springer, Berlin, 1997.

[31] A. Bortfeldt and H. Gehring. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131(1):143–161, 2001.

[32] A. Bortfeldt, H. Gehring, and D. Mack. A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*, 29(5):641–662, 2003.

[33] M. A. Boschetti. New lower bounds for the three-dimensional finite bin packing problem. *Discrete Applied Mathematics*, 140(1-3):241–258, 2004.

[34] M. F. Brameier and W. Banzhaf. *Linear Genetic Programming (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[35] D. J. Brown, B. S. Baker, and H. P. Katseff. Lower bounds for on-line two-dimensional packing algorithms. *Acta Informatica*, 18(2):1982, 1982.

[36] L. Brunetta and P. Grégoire. A general purpose algorithm for three-dimensional packing. *INFORMS J. on Computing*, 17(3):328–338, 2005.

[37] M. J. Brusco, G. M. Thompson, and L. W. Jacobs. A morph-based simulated annealing heuristic for a modified bin-packing problem. *Journal of the Operational Research Society*, 48(4):433–439, 1997.

[38] E. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research*, 55(4):655–671, 2004.

[39] E. Burke, G. Kendall, and G. Whitwell. A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock cutting problem. *INFORMS Journal On Computing (accepted)*, to appear 2009.

[40] E. Burke and J. Newell. Solving examination timetabling problems through adaptation of heuristic orderings. *Annals of Operations Research*, 129:107–134, 2004.

[41] E. K. Burke, Y. Bykov, J. Newall, and S. Petrovic. A time-predefined approach to course timetabling. *Yugoslav Journal of Operational Research*, 13(2):139–151, 2003.

[42] E. K. Burke, Y. Bykov, J. P. Newall, and S. Petrovic. A time-predefined local search approach to exam timetabling problems. *IIE Transactions on Operations Engineering*, 36(6):509–528, 2004.

[43] E. K. Burke, P. Cowling, and J. D. L. Silva. Hybrid populationbased metaheuristic approaches for the space allocation problem. In *Proceedings of the IEEE Conference on Evolutionary Computation (CEC'01)*, pages 232–239, Seoul, Korea, May 2001.

[44] E. K. Burke, P. Cowling, J. D. L. Silva, and B. McCollum. Three methods to automate the space allocation process in uk universities. In E. K. Burke and W. Erben, editors, *Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling (PATAT 2000)*, pages 254–272, Konstanz, Germany, August 2000.

[45] E. K. Burke, P. Cowling, J. D. L. Silva, and S. Petrovic. Combining hybrid metaheuristics and populations for the multiobjective optimisation of space allocation problems. In *Proceeedings of the ACM Genetic and Evolutionary Computation Conference (GECCO'01)*, pages 1252–1259, San Francisco, CA, USA, July 2001.

[46] E. K. Burke, M. Dror, S. Petrovic, and R. Qu. Hybrid graph heuristics within a hyperheuristic approach to exam timetabling problems. In *The Next Wave in Computing, Optimization, and Decision Technologies. Conference Volume of the 9th informs Computing Society Conference*, pages 79–91, Annapolis, Maryland, USA, January 2005.

[47] E. K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Schulenburg. Hyperheuristics: An emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Meta-Heuristics*, pages 457–474. Kluwer, Boston, Massachusetts, 2003.

[48] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In T. Runarsson, H.-G. Beyer, E. Burke, J. J.Merelo-Guervos, D. Whitley, and X. Yao, editors, *LNCS 4193, Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN'06)*, pages 860–869, Reykjavik, Iceland, September 2006.

[49] 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.

In *Proceedings of the 9th ACM Genetic and Evolutionary Computation Conference (GECCO'07)*, pages 1559–1565, London, UK., July 2007.

[50] E. K. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Kluwer, Boston, 2005.

[51] E. K. Burke, G. Kendall, J. D. Landa Silva, R. F. J. O'Brien, and E. Soubeiga. An ant algorithm hyperheuristic for the project presentation scheduling problem. In *Proceedings of the Congress on Evolutionary Computation 2005 (CEC'05)*, volume 3, pages 2263–2270, Edinburgh, U.K., September 2005.

[52] E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyper-heuristic for timetabling and rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[53] E. K. Burke, B. MacCarthy, S. Petrovic, and R. Qu. Knowledge discovery in a hyper-heuristic using case-based reasoning on course timetabling. In *LNCS 2740, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT'02)*, pages 276–286, KaHo St.-Lieven, Gent, Belgium, August 2002.

[54] E. K. Burke, B. MacCarthy, S. Petrovic, and R. Qu. Multiple-retrieval case based reasoning for course timetabling problems. *Journal of the Operational Research Society*, 57(2):148–162, 2006.

[55] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu. A graph-based hyper heuristic for timetabling problems. *European Journal of Operational Research*, 176:177–192, 2007.

[56] E. K. Burke, S. Petrovic, and R. Qu. Case-based heuristic selection for timetabling problems. *J. of Scheduling*, 9(2):115–132, 2006.

[57] A. Caprara and M. Monaci. On the two-dimensional knapsack problem. *Oper. Res. Lett.*, 32(1):5–14, 2004.

[58] F. Castillo, A. Kordon, G. Smits, B. Christenson, and D. Dickerson. Pareto front genetic programming parameter selection based on design of experiments and industrial data. In M. K. et al., editor, *Proceedings of the 8th annual conference on Genetic and*

evolutionary computation (GECCO'06), pages 1613–1620, Seattle, Washington, USA, July 2006.

[59] K. Chakhlevitch and P. I. Cowling. Choosing the fittest subset of low level heuristics in a hyperheuristic framework. In *LNCS 3448, Evolutionary Computation in Combinatorial Optimization, 5th European Conference, (EvoCOP'05)*, pages 23–33, Lausanne, Switzerland, 2005.

[60] B. Chazelle. The bottom-left bin packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32(8):697–707, 1983.

[61] S.-H. Chen and C.-C. Liao. Agent-based computational modeling of the stock pricevolume relation. *Information Sciences*, 170(1):75–100, 2005.

[62] N. Christofides and E. Hadjiconstantinou. An exact algorithm for orthogonal 2d cutting problems using guillotine cuts. *European Journal of Operational Research*, 83:21–38, 1995.

[63] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25(1):30–44, 1977.

[64] P. C. Chu and J. E. Beasley. A genetic algorithm for the generalised assignment problem. *Computers and Operations Research*, 24(1):17–23, 1997.

[65] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.

[66] C. K. Chua, V. Narayanan, and J. Loh. Constraint-based spatial representation technique for the container packing problem. *Integrated Manufacturing Systems*, 9(1):23–33, 1998.

[67] G. F. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier. algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191(1):61–85, 2008.

[68] E. G. Coffman Jr, C. Courcoubetis, M. Garey, D. S. Johnson, P. W. Shor, R. R. Weber, and M. Yannakakis. Bin packing with discrete item sizes, part i: Perfect

packing theorems and the average case behavior of optimal packings. *SIAM J. Disc. Math.*, 13:384–402, 2000.

[69] E. G. Coffman Jr, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Combinatorial analysis. In D. Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*. Kluwer, 1998.

[70] E. G. Coffman Jr, D. S. Johnson, P. W. Shor, and R. R. Weber. Bin packing with discrete item sizes, part ii: Tight bounds on first fit. *Random Structures and Algorithms*, 10:69–101, 1997.

[71] E. G. Coffman Jr. and J. C. Lagarias. Algorithms for packing squares: A probabilistic analysis. *SIAM Journal on Computing*, 18:166–185, 1989.

[72] A. L. Corcoran and R. L. Wainwright. A genetic algorithm for packing in three dimensions. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pages 1021–1030, New York, NY, USA, 1992.

[73] P. Cowling and K. Chakhlevitch. Hyperheuristics for managing a large collection of low level heuristics to schedule personnel. In *Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC03)*, pages 1214–1221, Canberra, Australia, December 2003.

[74] P. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In E. K. Burke and W. Erben, editors, *Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling (PATAT 2000)*, pages 176–190, Konstanz, Germany, August 2001.

[75] P. Cowling, G. Kendall, and E. Soubeiga. A parameter-free hyperheuristic for scheduling a sales summit. In *Proceedings of 4th Metahuristics International Conference (MIC 2001)*, pages 127–131, Porto, Portugal, July 2001.

[76] P. Cowling, G. Kendall, and E. Soubeiga. Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In S. Cagoni, J. Gottlieb, E. Hart, M. Middendorf, and R. Gnther, editors, *LNCS 2279, Applications of Evolutionary Computing : Proceedings of Evo Workshops 2002*, pages 1–10, Kinsale, Ireland, April 2002. Springer-Verlag.

[77] P. I. 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, CEC '02)*, pages 1185–1190, May 2002.

[78] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, July 1985.

[79] A. Cuesta-Canada, L. Garrido, and H. Terashima-Marin. Building hyper-heuristics through ant colony optimization for the 2d bin packing problem. In *LNCS 3684, Proceedings of the 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, pages 654–660, Melbourne, Australia, September 2005.

[80] V. D. Cung, M. Hifi, and B. Le Cun. Constrained two-dimensional cutting stock problemsa best-first branch-and-bound algorithm. *International Transactions in Operational Research*, 7(3):185–210, 2000.

[81] C. H. Dagli and A. Hajakbari. Simulated annealing approach for solving stock cutting problem. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 221–223, Los Angeles, CA, USA, 1990.

[82] J. M. Daida, J. D. Hommes, T. F. Bersano-Begey, S. J. Ross, and J. F. Vesecky. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21, pages 417–442. MIT Press, Cambridge, MA, USA, 1996.

[83] E. den Boef, J. Korst, S. Martello, D. Pisinger, and D. Vigo. Erratum to "the three-dimensional bin packing problem": Robot-packable and orthogonal variants of packing problems. *Oper. Res.*, 53(4):735–736, 2005.

[84] C. Dimopoulos and A. M. S. Zalzala. Investigating the use of genetic programming for a classic one-machine scheduling problem. *Advances in Engineering Software*, 32(6):489–498, 2001.

[85] U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22:25–40, 1995.

[86] K. Dowsland. Some experiments with simulated annealing techniques for packing problems. *European Journal of Operational Research*, 68(3):389–399, 1993.

[87] K. Dowsland, E. Soubeiga, and E. K. Burke. A simulated annealing hyper-heuristic for determining shipper sizes. *European Journal of Operational Research*, 179(3):759–774, 2007.

[88] K. A. Dowsland and W. Dowsland. Packing problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[89] H. Dyckhoff. A new linear programming approach to the cutting stock problem. *Operations Research*, 29(6):1092–1104, 1981.

[90] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[91] K. Eisemann. The trim problem. *Management Science*, 3(3):279–284, 1957.

[92] M. Eley. Solving container loading problems by block arrangement. *European Journal of Operational Research*, 141(2):393–409, 2002.

[93] D. I. Ellis, D. Broadhurst, and R. Goodacre. Rapid and quantitative detection of the microbial spoilage of beef by fourier transform infrared spectroscopy and machine learning. *Analytica Chimica Acta*, 514(2):193–201, 2004.

[94] E. Ersoy, E. Ozcan, and S. Uyar. Memetic algorithms and hyperhill-climbers. In *Proceedings of the 3rd Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA'07)*, pages 159–166, Paris, France, August 2007.

[95] L. Faina. An application of simulated annealing to the cutting stock problem. *European Journal of Operational Research*, 114(3):542–556, 1999.

[96] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996.

[97] E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. In *Proceedings of the IEEE 1992 Int. Conference on Robotics and Automation*, pages 1186–1192, Nice, France, May 1992.

[98] H. L. Fang, P. Ross, and D. Corne. A promising genetic algorithm approach to job shop scheduling, rescheduling, and open-shop scheduling problems. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA'93)*, pages 375–382, University of Illinois, Urbana-Champaign, July 1993.

[99] H.-L. Fang, P. Ross, and D. Corne. A promising hybrid ga/heuristic approach for open shop scheduling problems. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI '94)*, pages 590–594, 1994.

[100] T. Fanslau and A. Bortfeldt. A tree search algorithm for solving the container loading problem. *INFORMS Journal on Computing (to appear, available online July)*, 2009.

[101] O. Faroe, D. Pisinger, and M. Zachariasen. Guided local search for the three-dimensional bin-packing problem. *INFORMS Journal on Computing*, 15(3):267–283, 2003.

[102] S. Fekete and J. Schepers. A new exact algorithm for general orthogonal d-dimensional knapsack problems. In *Algorithms (ESA 97). Springer Lecture Notes in Computer Science*, volume 1284, pages 144–156, Springer, Berlin, 1997.

[103] S. Fekete, J. Schepers, and J. V. der Veen. An exact algorithm for higher-dimensional orthogonal packing. *Operations Research*, 55(3):569–587, 2007.

[104] C. E. Ferreira, F. K. Miyazawa, and Y. Wakabayashi. Packing of squares into squares. *Pesquisa Operacional*, 19:223–237, 1999.

[105] E. Fink. How to solve it automatically: selection among problem-solving methods. In *Proceedings of the Fourth International Conference of AI Planning Systems*, pages 128–136, Pittsburgh, Pennsylvania, USA, June 1998.

[106] H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In *Factory Scheduling Conference*, pages 10–12, Carnegie Institue of Technology, May 1961.

[107] H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job shop scheduling rules. *Industrial Scheduling*, pages 225–251, 1963.

[108] K. Fleszar and K. S. Hindi. New heuristics for one-dimensional bin-packing. *Computers and Operations Research*, 29(7):821–839, 2002.

[109] F. D. Francone and L. M. Deschaine. Getting it right at the very start – building project models where data is expensive by combining human expertise, machine learning and information theory. In *Proceedings of the Business and Industry Symposium*, Washington, DC, April 2004.

[110] C. Fujiko and J. Dickinson. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In *Proceedings of the 2nd International Conference on Genetic Algorithms and their application*, pages 236–240, Cambridge, Massachusetts, United States, 1987. L. Erlbaum Associates Inc.

[111] S. Fujita and M. Yamashita. Approximation algorithms for multiprocessor scheduling problem. *IEICE Transactions on Information and Systems*, 83(3):503–509, 2000.

[112] A. S. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In *Eighteenth national conference on Artificial intelligence*, pages 641–648, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[113] A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *LNCS 3103. Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO '04)*, pages 483–494, Seattle, WA, USA, 2004. Springer-Verlag.

[114] A. S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation (MIT Press)*, 16(1):31–1, 2008.

[115] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-Completeness*. W.H. Freeman and Company, San Fransisco, 1979.

[116] H. Gehring and A. Bortfeldt. A genetic algorithm for solving the container loading problem. *International Transactions in Operational Research*, 4(5):401–418, 1997.

[117] M. Gehring, K. Menscher, and M. Meyer. A computer-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 44(2):277–288, 1990.

[118] C. D. Geiger, R. Uzsoy, and H. Aytug. Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *Journal of Scheduling*, 9(1):7–34, 2006.

[119] J. A. George and D. F. Robinson. A heuristic for packing boxes into a container. *Computers and Operations Research*, 7(3):147–156, 1980.

[120] P. Gilmore and R. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[121] P. Gilmore and R. Gomory. A linear programming approach to the cutting-stock problem - part ii. *Operations Research*, 11:863–888, 1963.

[122] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operational Research*, 13(5):533–549, 1986.

[123] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[124] J. Gratch, S. Chein, and G. DeJong. Learning search control knowledge for deep space network scheduling. In *Proceedings of the 10th International Conference on Machine Learning*, pages 135–142, Amherst, MA, June 1993.

[125] J. Gratch and S. A. Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Artif. Intell. Res.*, 4:365–396, 1996.

[126] J. Gratch and G. DeJong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 235–240, San Jose, California, July 1992.

[127] J. N. D. Gupta and J. C. Ho. A new heuristic algorithm for the one-dimensional bin-packing problem. *Production Planning and Control*, 10(6):598–603, 1999.

[128] E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

[129] E. Hadjiconstantinou and M. Iori. A hybrid genetic algorithm for the two-dimensional single large object placement problem. *European Journal of Operational Research*, 183(3):1150–1166, 2007.

[130] R. W. Haessler. A heuristic programming solution to a nonlinear cutting stock problem. *Management Science*, 17(12):793–802, 1971.

[131] R. W. Haessler and P. E. Sweeney. Cutting stock problems and solution procedures. *European Journal of Operational Research*, 54(2):141–150, 1991.

[132] L. Han and G. Kendall. Guided operators for a hyper-heuristic genetic algorithm. In *Proceedings of The 16th Australian Conference on Artificial Intelligence (AI'03)*, pages 807–820, Perth, Australia, December 2003.

[133] L. Han and G. Kendall. Investigation of a tabu assisted hyper-heuristic genetic algorithm. In *Proceedings of Congress on Evolutionary Computation (CEC'03)*, volume 3, pages 2230–2237, Canberra, Australia, December 2003.

[134] L. Han, G. Kendall, and P. Cowling. An adaptive length chromosome hyperheuristic genetic algorithm for a trainer scheduling problem. In *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'02)*, pages 267–271, Orchid Country Club, Singapore, November 2002.

[135] P. Hansen and N. Mladenovic. An introduction to variable neighborhood search. In S. Voss, I. H. Osman, and C. Roucairol, editors, *Metaheuristics, advances and trends in local search paradigms for optimization*, pages 433–458. Kluwer, Norwell, MA, USA, 1999.

[136] E. Hart and P. Ross. A heuristic combination method for solving job-shop scheduling problems. In *LNCS 1498, Proceedings of the 5th International Conference on Parallel Problem Solving from Nature (PPSN'98)*, pages 845–854, Amsterdam, The Netherlands, September 1998.

[137] J. F. Hicklin. Application of the genetic algorithm to automatic program generation. Master's thesis, Dept. of Computer Science, University of Idaho, 1986.

[138] M. Hifi and V. Zissimopolous. Constrained two-dimensional cutting: An improvement of christofides and whitlocks exact algorithm. *Journal of the Operational Research Society*, 48:324–331, 1997.

[139] M. Hifi and V. Zissimopoulos. A recursive exact algorithm for weighted two-dimensional cutting. *European Journal of Operational Research*, 91(3):553–564, 1996.

[140] N. B. Ho and J. C. Tay. Evolving dispatching rules for solving the flexible job-shop problem. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'05)*, pages 2848–2855, Edinburgh, UK, September 2005.

[141] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, First published by University of Michigan Press 1975, 1992.

[142] O. Holthaus. Decomposition approaches for solving the integer one-dimensional cutting stock problem with different types of standard lengths. *European Journal of Operational Research*, 141(2):295–312, 2002.

[143] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34–57, 2001.

[144] W. Huang and K. He. A new heuristic algorithm for cuboids packing with no orientation constraints. *Computers and Operations Research*, 36(2):425–432, 2009.

[145] R. Hübscher and F. Glover. Applying tabu search with influential diversification to multiprocessor scheduling. *Computers and Operations Research*, 21:877–884, 1994.

[146] S. M. Hwang, C. Y. Kao, and J. T. Horng. On solving rectangle bin packing problems using genetic algorithms. In *Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1583–1590, IEEE, San Antonio, TX., 1994.

[147] A. Ieumwananonthachai. *Automated Design of Knowledge-Lean Heuristics: Learning, Resource Scheduling, and Generalization*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[148] A. Ieumwananonthachai, A. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan. Intelligent process mapping through systematic improvement of heuristics. *Journal of Parallel and Distributed Computing*, 15(2):118–142, 1992.

[149] A. Ieumwananonthachai and B. W. Wah. Teacher: A genetics-based system for learning and for generalizing heuristics. In X. Yao, editor, *Evolutionary Computation*, pages 124–170. World Scientific Publishing Co. Pte. Ltd., 1999.

[150] N. Ivancic, K. Mathur, and B. Mohanty. An integer-programming based heuristic approach to the three-dimensional packing problem. *Journal of Manufacturing and Operations Management*, 2:268–298, 1989.

[151] D. Jakobovic, L. Jelenkovic, and L. Budin. Genetic programming heuristics for multiple machine scheduling. In *LNCS 4445. Proceedings of the European Conference on Genetic Programming (EUROGP'07)*, pages 321–330, Valencia, Spain, April 2007.

[152] S. Jakobs. On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88(1):165–181, 1996.

[153] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packaging algorithms. *SIAM Journal on Computing*, 3(4):299–325, December 1974.

[154] D. Joslin and D. P. Clements. Squeaky wheel optimization. *Journal of Artificial Intelligence Research*, 10:353–373, 1999.

[155] M. Kaboudan. A measure of time series predictability using genetic programming applied to stock returns. *Journal of Forecasting*, 18(5):345–357, 1999.

[156] T. Kampke. Simulated annealing: use of new tool in bin packing. *Ann. Oper. Res.*, 16(1-4):327–332, 1988.

[157] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.

[158] K. Karabulut and M. M. Inceoglu. A hybrid genetic algorithm for packing in 3d with deepest bottom left with fill method. In *LNCS 3261. Advances in Information Systems*, pages 441–450, October 2004.

[159] R. E. Keller and R. Poli. Linear genetic programming of parsimonious metaheuristics. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'07)*, pages 4508–4515, Singapore, September 2007.

[160] G. Kendall and M. Hussin. A tabu search hyper-heuristic approach to the examination timetabling problem at the mara university of technology. In E. Burke and T. Trick, editors, *LNCS 3616. Revised Selected Papers of the 5th International Conference of Practice and Theory of Automated Timetabling V (PATAT'04)*, pages 270–293, Pittsburgh, USA, 2005.

[161] G. Kendall and N. Mohd Hussin. An investigation of a tabu search based hyper-heuristic for examination timetabling. In *Multi-disciplinary Scheduling: Theory and Applications (eds. G. Kendall, E. K. Burke, S.Petrovic and M. Gendreau) Selected papers from the 1st International Conference on Multi-disciplinary Scheduling: Theory and Applications (MISTA'05)*, pages 309–328, Nottingham, UK, 2005.

[162] C. Kenyon. Best-fit bin-packing with random order. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 359–364, 1996.

[163] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 768–774, San Francisco, CA, USA, August 1989.

[164] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection.* The MIT Press, Boston, Massachusetts, 1992.

[165] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs.* The MIT Press, Cambridge, Massachusetts, 1994.

[166] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem solving.* Morgan Kaufmann, San Francisco, 1999.

[167] J. R. Koza and R. Poli. Genetic programming. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 127–164. Kluwer, Boston, 2005.

[168] N. Krasnogor. *Studies on the Theory and Design Space of Memetic Algorithms.* PhD thesis, University of the West of England, 2002.

[169] N. Krasnogor. Self generating metaheuristics in bioinformatics: The proteins structure comparison case. *Genetic Programming and Evolvable Machines*, 5(2):181–201, 2004.

[170] N. Krasnogor and S. Gustafson. A study on the use of "self-generation" in memetic algorithms. *Natural Computing*, 3(1):53–76, 2004.

[171] B. Kröger. Guillotinable bin packing: A genetic approach. *European Journal of Operational Research*, 84(3):645–661, 1995.

[172] R. Kumar, A. H. Joshi, K. K. Banka, and P. I. Rockett. Evolution of hyperheuristics for the biobjective 0/1 knapsack problem by multiobjective genetic programming. In *Proceedings of the 10th ACM conference on Genetic and evolutionary computation (GECCO'08)*, pages 1227–1234, Atlanta, GA, USA, 2008.

[173] R. Kumar, B. Kumar Bal, and P. I. Rockett. Multiobjective genetic programming approach to evolving heuristics for the bounded diameter minimum spanning tree problem. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO '09)*, pages 309–316, Montreal, Canada, July 2009.

[174] K. Lagus, I. Karanta, and J. Yla-Jaaski. Paginating the generalized newspaper, a comparison of simulated annealing and a heuristic method. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature (PPSN '96)*, pages 549–603, Berlin, 1996.

[175] K. K. Lai and J. W. M. Chan. Developing a simulated annealing algorithm for the cutting stock problem. *Computers and Industrial Engineering*, 32(1):115–127, 1997.

[176] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pan, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 1997.

[177] W. B. Langdon and R. Poli. Evolutionary solo pong players. In D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary*

Computation, volume 3, pages 2621–2628, Edinburgh, UK, September 2005. IEEE Press.

[178] H. F. Lee and E. C. Sewell. The strip-packing problem for a boat manufacturing firm. *IIE Transactions*, 31(7):639–651, 1999.

[179] N. Lesh, J. Marks, and M. McMahon, A.and Mitzenmacher. New heuristic and interactive approaches to 2d rectangular strip packing. *Journal of Experimental Algorithmics*, 10:1–18, 2005.

[180] J. Y. T. Leung, T. W. Tam, C. S. Wong, G. H. Young, and F. Y. L. Chin. Packing squares into a square. *Journal of Parallel and Distributed Computing*, 10:271–275, 1990.

[181] D. R. Lewin, S. Lachman-Shalem, and B. Grosman. The role of process system engineering (PSE) in integrated circuit (IC) manufacturing. *Control Engineering Practice*, 15(7):793–802, July 2006.

[182] K. Li and K. H. Cheng. On three dimensional packing. *SIAM Journal on Computing*, 19:847–867, 1990.

[183] K. Li and K. H. Cheng. Heuristic algorithms for on-line packing in three dimensions. *Journal of Algorithms*, 13:589–605, 1992.

[184] A. Lim, B. Rodrigues, and Y. Wang. A multi-faced buildup algorithm for three-dimensional packing problems. *OMEGAInternational Journal of Management Science*, 31(6):471–481, 2003.

[185] A. Lim and X. Zhang. The container loading problem. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC'05)*, pages 913–917, New York, NY, USA, 2005. ACM Press.

[186] J. L. Lin, B. Foote, S. Pulat, C. H. Chang, and J. Y. Cheung. Hybrid genetic algorithm for container packing in three dimensions. In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*, pages 353–359, Orlando, FL, USA, March 1993.

[187] D. Liu and H. Teng. An improved bl-algorithm for genetic algorithms of the orthogonal packing of rectangles. *European Journal of Operational Research*, 112(2):413–419, 1999.

[188] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[189] A. Lodi, S. Martello, and D. Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. on Computing*, 11(4):345–357, 1999.

[190] A. Lodi, S. Martello, and D. Vigo. Heuristic algorithms for the three-dimensional bin packing problem. *European Journal of Operational Research*, 141(2):410–420, 2002.

[191] H. T. Loh and A. Y. C. Nee. A packing algorithm for hexahedral boxes. In *Proceedings of the Industrial Automation Conference*, pages 115–126, Singapore, 1992.

[192] K.-H. Loh, B. Golden, and E. Wasil. Solving the one-dimensional bin packing problem with a weight annealing heuristic. *Computers and Operations Research*, 35(7):2283–2291, 2008.

[193] J. Lohn, G. Hornby, and D. Linden. Evolutionary antenna design for a NASA spacecraft. In U.-M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, May 2004.

[194] J. H. Lorie and L. J. Savage. Three problems in capital rationing. *The Journal of Business*, 28:229–239, 1955.

[195] K. A. Marko and R. J. Hampo. Application of genetic programming to control of vehicle systems. In *Proceedings of the Intelligent Vehicles Symposium*, pages 191–195, Detroit, MI, USA, July 1992.

[196] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS J. on Computing*, 15(3):310–319, 2003.

[197] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operat. Res.*, 48:256–267, 2000.

[198] S. Martello, D. Pisinger, D. Vigo, E. D. Boef, and J. Korst. Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Softw.*, 33(1), 2007.

[199] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, Chichester, 1990.

[200] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

[201] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44(3):388–399, 1998.

[202] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[203] A. Meir and L. Moser. On packing of squares and cubes. *Journal of Combinatorial Theory*, 5:126–134, 1968.

[204] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5):525–530, 1978.

[205] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 2, pages 1135–1142, Orlando, Florida, USA, July 1999. Morgan Kaufmann.

[206] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.

[207] F. K. Miyazawa and Y. Wakabayashi. An algorithm for the threedimensional packing problem with asymptotic performance analysis. *Algorithmica*, 18(1):122–144, 1997.

[208] F. K. Miyazawa and Y. Wakabayashi. Approximation algorithms for the orthogonal z-oriented three-dimensional packing problem. *SIAM Journal on Computing*, 29:1008–1029, 1999.

[209] F. K. Miyazawa and Y.Wakabayashi. Cube packing. *Theoretical Computer Science*, 297(1-3):355–366, 2003.

[210] A. Moura and J. F. Oliveira. A grasp approach to the container-loading problem. *IEEE Intelligent Systems*, 20(4):50–57, 2005.

[211] B. K. A. Ngoi, M. L. Tay, and E. S. Chua. Applying spatial representation techniques to the container packing problem. *International Journal of Production Research*, 32(1):111–123, 1994.

[212] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[213] M. Oltean and D. Dumitrescu. Evolving TSP heuristics using multi expression programming. In M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *LNCS 3037. Proceedings of the 4th International Conference on Computational Science (ICCS'04)*, pages 670–673, Krakow, Poland, June 2004.

[214] M. Oltean and C. Grosan. Evolving evolutionary algorithms using multi expression programming. In W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, and J. Ziegler, editors, *LNAI 2801. Proceedings of the 7th European Conference on Artificial Life*, pages 651–658, Dortmund, Germany, September 2003.

[215] M. O'Neill, R. Cleary, and N. Nikolov. Solving knapsack problems with attribute grammars. In *Proceedings of the Third Grammatical Evolution Workshop (GEWS'04)*, Seattle, WA, USA., 2004.

[216] E. Özcan, B. Bilgin, and E. E. Korkmaz. Hill climbers and mutational heuristics in hyperheuristics. In T. Runarsson, H.-G. Beyer, E. Burke, J. J.Merelo-Guervos, D. Whitley, and X. Yao, editors, *LNCS 4193, Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006)*, pages 202–211, Reykjavik, Iceland, September 2006.

[217] G. L. Pappa and A. A. Freitas. Automatically evolving rule induction algorithms tailored to the prediction of postsynaptic activity in proteins. *Intelligent Data Analysis*, 13(2):243–259, 2009.

[218] T. Perkis. Stack-based genetic programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 148–153, Orlando, Florida, USA, June 1994.

[219] D. Pisinger. *Algorithms for Knapsack Problems. Report 95/1.* PhD thesis, University of Copenhagen, 1995.

[220] D. Pisinger. Heuristics for the container loading problem. *European Journal of Operational Research*, 141(2):382–392, 2002.

[221] D. Pisinger and J. Egeblad. Heuristic approaches for the two and three dimensional knapsack packing problems, technical report no.06-13. Technical report, University of Copenhagen, Dept of Computer Science, 2006.

[222] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34(8):2403–2435, 2007.

[223] D. Pisinger and P. Toth. Knapsack problems. In D. Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1–89. Kluwer, 1998.

[224] R. Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Technical Report CSRP-96-14, University of Birmingham, School of Computer Science, Aug. 1996.

[225] R. Poli. Parallel distributed genetic programming. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England, 1999.

[226] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, pages 211–223, Essex, April 2003. Springer-Verlag.

[227] R. Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the 2nd Annual Conference*, pages 278–285, July 1997.

[228] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming.* lulu.com, freely available at http://www.gp-field-guide.org.uk, 2008.

[229] R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1), March 2003.

[230] R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.

[231] R. Poli, N. F. McPhee, and J. E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004.

[232] R. Poli, J. R. Woodward, and E. K. Burke. A histogram-matching approach to the evolution of bin-packing strategies. In *Proceedings of the Congress on Evolutionary Computation (CEC 2007)*, pages 3500–3507, Singapore, September 2007.

[233] A. Ramesh Babu and N. Ramesh Babu. Effective nesting of rectangular parts in multiple rectangular sheets using genetic and heuristic algorithms. *International Journal of Production Research*, 37(7):1625–1643, 1999.

[234] R. L. Rao and S. S. Iyengar. Bin-packing by simulated annealing. *Computers and Mathematics with Applications*, 27(5):71–82, 1994.

[235] P. Rattadilok, A. Gaw, and R. Kwan. Distributed choice function hyper-heuristics for timetabling and scheduling. In E. Burke and M.Trick, editors, *Practice and Theory of Automated Timetabling V, Springer Lecture notes in Computer Science*, volume 3616, pages 51–67, 2005.

[236] W. T. Rhee and M. Talagrand. On line bin packing with items of random size. *Math. Oper. Res.*, 18:438–445, 1993.

[237] M. B. Richey. Improved bounds for harmonic-based bin packing algorithms. *Discrete Applied Mathematics*, 34:203–227, 1991.

[238] F. Rinaldi and A. Franz. A two-dimensional strip cutting problem with sequencing constraint. *European Journal of Operational Research*, 183(3):1371–1384, 2007.

[239] P. Ross. Hyper-heuristics. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 529–556. Kluwer, Boston, 2005.

[240] P. Ross, E. Hart, and D. Corne. Some observations about ga-based exam timetabling. In *Selected papers from the Second International Conference on Practice and Theory of Automated Timetabling (PATAT'97)*, pages 115–129, London, UK, 1998.

[241] P. Ross, J. G. Marin-Blazquez, S. Schulenburg, and E. Hart. Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyperheurstics. In *Proceedings of the Genetic and Evolutionary Computation Conference 2003 (GECCO '03)*, pages 1295–1306, Chicago, Illinois, 2003.

[242] P. Ross, S. Schulenburg, J. G. Marin-Blazquez, and E. Hart. Hyper heuristics: learning to combine simple heuristics in bin packing problems. In *Proceedings of the Genetic and Evolutionary Computation Conference 2002 (GECCO '02)*, New York, NY., 2002.

[243] A. Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[244] K. Schimmelpfeng and S. Helber. Application of a real-world university-course timetabling model solved by integer programming. *OR Spectrum*, 29(4):783–803, 2007.

[245] W. Schneider. Trim-loss minimization in a crepe-rubber mill; optimal solution versus heuristic in the 2 (3) - dimensional case. *European Journal of Operational Research*, 34(3):249–412, 1988.

[246] A. Scholl, R. Klein, and C. Jurgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research*, 24(7):627–645, 1997.

[247] S. R. Schwartz and B. W. Wah. Automated parameter tuning in stereo vision under time constraints. In *Proceedings of the 4th International Conference on Tools with Artificial Intelligence (TAI'92)*, pages 162–169, Arlington, VA, USA, November 1992.

[248] P. Schwerin and G. Wäscher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4(5):377–389, 1997.

[249] S. Seiden, R. Van Stee, and L. Epstein. New bounds for variable-sized online bin packing. *SIAM Journal on Computing*, 32(2):455–469, 2003.

[250] J. D. L. Silva. *Metaheuristic and Multiobjective Approaches for Space Allocation*. PhD thesis, Department of Computer Science, Nottingham University, 2003.

[251] Z. Sinuany-Stern and I. Weiner. The one dimensional cutting stock problem using two objectives. *The Journal of the Operational Research Society*, 45(2):231–236, 1994.

[252] E. Soubeiga. *Development and Application of Hyperheuristics to Personnel scheduling*. PhD thesis, Univesity of Nottingham, School of Computer Science, 2003.

[253] T. Soule and R. B. Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283–309, 2002.

[254] L. Spector, H. Barnum, and H. J. Bernstein. Genetic programming for quantum computers. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the 3rd Annual Conference*, pages 365–373, University of Wisconsin, Madison, Wisconsin, USA, July 1998.

[255] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy. Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 2239–2246, Washington D.C., USA, July 1999.

[256] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992.

[257] R. H. Storer, S. D. Wu, and R. Vaccari. Problem and heuristic space search strategies for job shop scheduling. *ORSA Journal on Computing*, 7(4):453–467, 1995.

[258] W. A. Tackett. Genetic generation of "dendritic" trees for image classification. In *Proceedings of the World Conference on Neural Networks*, pages 646–649, Portland, Oregon, USA, July 1993. IEEE Press.

[259] J. Tavares, P. Machado, A. Cardoso, F. B. Pereira, and E. Costa. On the evolution of evolutionary algorithms. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *LNCS 3003. Proceedings of the European Conference on Genetic Programming (EUROGP'04)*, pages 389–398, Coimbra, Portugal, April 2004.

[260] J. C. Tay and N. B. Ho. Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers and Industrial Engineering*, 54(3):453–473, 2008.

[261] C. C. Teng and B. W. Wah. An automated design system for finding the minimal configuration of a feed-forward neural network. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1295–1300, June 1994.

[262] H. Terashima-Marin, E. J. Flores-Alvarez, and P. Ross. Hyper-heuristics and classifier systems for solving 2d-regular cutting stock problems. In *Proceeedings of the ACM Genetic and Evolutionary Computation Conference (GECCO'05)*, pages 637–643, Washington, D.C. USA, June 2005.

[263] H. Terashima-Marin, P. M. Ross, and M. Valenzuela. Evolution of constraint satisfaction strategies in examination timetabling. In W. Banzhaf, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, pages 635–642, Orlando, Florida, USA, July 1999. Morgan Kaufmann.

[264] H. Terashima-Marin, C. J. F. Zarate, P. Ross, and M. Valenzuela-Rendon. A ga-based method to produce generalized hyper-heuristics for the 2d-regular cutting stock problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 591–598, Seattle, WA, USA, July 2006.

[265] H. Terashima-Marin, C. J. F. Zarate, P. Ross, and M. Valenzuela-Rendon. Comparing two models to generate hyper-heuristics for the 2d-regular bin-packing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, pages 2182–2189, London, England, July 2007.

[266] E. Tsang, P. Yung, and J. Li. EDDIE-automation, a decision support tool for financial forecasting. *Decision Support Systems*, 37(4):559–565, 2004.

[267] C. L. Valenzuela and P. Y. Wang. Heuristics for large strip packing problems with guillotine patterns: An empirical study. In *Proceedings of the Metaheuristics International Conference 2001 (MIC'01)*, pages 417–421, University of Porto, Porto, Portugal, 2001.

[268] J. M. Valerio de Carvalho. A note on branch-and-price algorithms for the one-dimensional cutting stock problems. *Computational Optimization and Applications*, 21(3):339–340, 2002.

[269] R. van Stee. An approximation algorithm for square packing. *Operations Research Letters*, 32(6):535–539, 2004.

[270] P. H. Vance. Branch-and-price algorithms for the one-dimensional cutting stock problem. *Computational Optimization and Applications*, 9(3):211–228, 1998.

[271] F. J. Vasko, F. E. Wolf, and K. L. Stott. A practical solution to a fuzzy two-dimensional cutting stock problem. *Fuzzy Sets and Systems*, 29(3):259–275, 1989.

[272] B. Vowk, A. S. Wait, and C. Schmidt. An evolutionary approach generates human competitive coreware programs. In M. Bedau, P. Husbands, T. Hutton, S. Kumar, and H. Sizuki, editors, *Workshop and Tutorial Proceedings 9th International Conference on the Simulation and Synthesis of Living Systems(Alife XI)*, pages 33–36, Boston, Massachusetts, September 2004.

[273] B. W. Wah. Population-based learning: A new method for learning from examples under resource constraints. *IEEE Trans. on Knowledge and Data Engineering*, 4(5):454–474, 1992.

[274] B. W. Wah, A. Ieumwananonthachai, L. C. Chu, and A. Aizawa. Genetics-based learning of new heuristics: Rational scheduling of experiments and generalization. *IEEE Trans. on Knowledge and Data Engineering*, 7(5):783–785, 1995.

[275] P. Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Operations Research*, 31(3):573–586, 1983.

[276] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[277] D. Whitley and J. P. Watson. Complexity and no free lunch. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 317–339. Kluwer, Boston, 2005.

[278] D. Whitley and J. P. Watson. Complexity theory and the no free lunch theorem. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Kluwer, Boston, 2005.

[279] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[280] M. H. Yang. An efficient algorithm to allocate shelf space. *European Journal of Operational Research*, 131(1):107–118, 2001.

[281] M. H. Yang and W. C. Chen. A study on shelf space allocation and management. *International Journal of Production Economics*, 60-61:309–317, 1999.

[282] A. C.-C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27(2):207–227, 1980.

[283] M. Yue. A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1$, for all l for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica (English Series)*, 7(4):321–331, 1991.

[284] D. Zhang, Y. Kang, and A. Deng. A new heuristic recursive algorithm for the strip rectangular packing problem. *Computers and Operations Research*, 33(8):2209–2217, 2006.