Evolving Bin Packing Heuristics with Genetic Programming

E.K. Burke, M.R. Hyde, and G. Kendall

School of Computer Science and Information Technology
The University of Nottingham
Nottingham, NG8 1BB
United Kingdom (UK)
mvh@cs.nott.ac.uk
http://cs.nott.ac.uk/∼mvh

Abstract. The bin-packing problem is a well known NP-Hard optimisation problem, and, over the years, many heuristics have been developed to generate good quality solutions. This paper outlines a genetic programming system which evolves a heuristic that decides whether to put a piece in a bin when presented with the sum of the pieces already in the bin and the size of the piece that is about to be packed. This heuristic operates in a fixed framework that iterates through the open bins, applying the heuristic to each one, before deciding which bin to use. The best evolved programs emulate the functionality of the human designed 'first-fit' heuristic. Thus, the contribution of this paper is to demonstrate that genetic programming can be employed to automatically evolve bin packing heuristics which are the same as high quality heuristics which have been designed by humans.

1 Introduction

The aim of this work is to explore how a genetic programming system might evolve good heuristics that can pack bins. The goal is to automatically generate good heuristics which can operate over a range of instances and, perhaps more importantly, do not require human intervention or parameter tuning. The aim is therefore to show that a genetic programming system is capable of generating heuristics which have the functionality of human-designed heuristics. In this work we use a basic genetic programming system which does not incorporate any of the more advanced techniques such as re-usable code, loops, or memory, all of which are explained in [1] and [2] for the interested reader.

1.1 Genetic Programming

Genetic programming [3] evolves a population of computer programs which are represented as tree structures. After each program (or individual) has been executed, its performance is assessed and a fitness rating is given to it. The genetic operators of crossover and reproduction are applied to individuals in the population in proportion to their relative fitness.

T.P. Runarsson et al. (Eds.): PPSN IX, LNCS 4193, pp. 860–869, 2006.

1.2 Hyper-heuristics

One of the aims of a hyper-heuristic [4,5] is to "raise the level of generality at which optimisation systems can operate" [5]. To this end, hyper-heuristics are heuristics which choose "between a set of low-level heuristics, using some learning mechanism" [6].

Of course, the No Free Lunch theorem [7,8] shows that all search algorithms have the same average performance on all problems defined on a given finite search space. 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 possible today. Indeed, research into hyper-heuristics is motivated by the assertion that in many real world problem solving environments, there are users who are interested in *"good-enough soon-enough cheap-enough"* solutions to their optimisation problems [5].

Some examples of hyper-heuristic methods are briefly explained here. Two hyper-heuristic methods have been tested on the one-dimensional bin-packing problem, a learning classifier system [9] and a genetic algorithm [10]. In [11], an ant algorithm hyper-heuristic chooses a sequence of low-level heuristics for the project presentation scheduling problem. A tabu search hyper-heuristic is applied in [12] to a nurse scheduling problem and a university course timetabling problem. A choice function has also been employed as a hyper-heuristic, to rank the low-level heuristics and choose the best [13]. A multi-objective hyper-heuristic is applied to space allocation and timetabling in [14]. A graph based hyper-heuristic is used for timetabling in [15]. Case based heuristic selection is applied on a timetabling problem in [16]. Finally, in [17], a simulated annealing hyper-heuristic is used to determine shipper sizes.

1.3 Using Genetic Programming as a Hyper-heuristic

This paper invesigates the role of genetic programming as a hyper-heuristic. The genetic programming system chooses between a set of low level building blocks to construct a heuristic which performs well in the environment given to it. In this case the building blocks are the function and terminal set shown in table 2, and the environment is the data set given to the system.

In previous work (e.g. [12]), hyper-heuristics have had their low-level heuristics given to them by the human programmer, and so the number and quality of heuristics that the hyper-heuristic has available is limited to those which a human can provide. The aim of this research is to show that even more of the decision process can be automated. For example, the human programmer would normally choose the low-level heuristics, from the space of all heuristics. This paper aims to show that by using a genetic programming system as a hyper-heuristic, *any* heuristic can be chosen from the space of all heuristics that can be constructed with the function and terminal set. The human programmer therefore need only supply the *potential* components of a heuristic.

2 The Bin Packing Problem

The one-dimensional bin-packing problem involves a set of integer-size pieces L, which must be packed into bins of a certain capacity C, using the minimum number of bins possible. In other words, the set of integers must be divided into the smallest number of subsets so that the sum of the sizes of the pieces in a subset does not exceed C [18].

2.1 The Problem

Twenty instances of the bin packing problem are used in this research, taken from benchmark data studied by Falkenauer in [19] and now maintained by Beasley in the OR-Library of Brunel University [20]. In each instance, there are 120 pieces, all uniformly distributed in (20, 100). These pieces are packed into bins of capacity 150 for every instance.

In this paper, the 'on-line' bin packing problem is studied. That is, we do not know in advance how many pieces there are or the size of those pieces. Our system must simply pack the pieces into the bins in the order they arrive, and the pieces cannot be moved once they have been placed in a bin [21]. This situation is likely to arise in the real world. For example when items come off a production line and are placed into containers, or packages of different sizes are packed into trucks at a depot, and only a certain number of trucks can be at the depot at any one time [21].

2.2 Existing Heuristics

The bin packing problem is known to be NP-Hard [22] so heuristics are commonly used to generate solutions that are of high enough quality for practical purposes, as a polynomial-time exact algorithm is unlikely to be found for the general case [21]. A number of examples of heuristics used in the online bin packing problem are described below:

Best Fit [23]: Puts the piece in the fullest bin that has room for it. Opens a new bin if the piece does not fit in any existing bin

Worst Fit [21]: 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 [21]: 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 [24]: Puts the piece in the right-most bin and opens a new bin if there is not enough room for it.

First Fit [24]: Puts the piece in the left-most bin that has room for it and opens a new bin if it does not fit in any open bin.

3 The Genetic Programming Hyper-heuristic

3.1 Evolving the Choice of Bin

Our system evolves a control program that decides whether to put a given piece into a given bin. An individual in the population is assessed by rating the result

created when the algorithm in Fig. 1 is run, where L = the list of all the pieces, and A = the array of bins. Note that when making the choice of whether to put the current piece in the current bin, each individual in the population is not constrained by whether it is legal to do so. For example, putting every piece in the first bin is permitted. However, this will lead to an illegal solution with a high penalty. For this reason, when the *best-of-run* individual produces a legal solution, it is because the system will have evolved an understanding of the rules, not because of artificial constraints imposed by humans.

```
For each piece p in L
  For each bin i in A
    output = evaluate(p, fullness of i, capacity of i)
    If (output > 0)
      place piece p in bin i
      break
    End If
  End For
End For
```

Fig. 1. Pseudo code showing the overall program structure

Initialisation Parameters. Table 1 shows the parameters of the run. No optimality is claimed for these parameters. They were chosen from a range of possible initialisation parameters, after a series of experiments, because they result in a good solution in reasonable time. The population size of 1000 was chosen because it gives a good range of solutions in the initial population, and it allows for reasonable run times. The maximum depth of the initial trees is a relatively low value because the maximum depth obtainable during the run is not limited. The 'Grow' method of initialising the trees [25] is used here because it creates an initial population of diverse structures. Using 50 as the maximum number of generations is a standard parameter used in [3], where generation 0 is the initial random population.

Fitness measure. The fitness measure is shown in equation 1, where:
B = Number of bins used, n = Number of pieces,
S_k = Size of piece k, and C = Bin capacity

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

This means that a fitness of zero is the perfect result, because the pieces are packed into the smallest number of bins possible. A fitness of 1000 is assigned to any illegal solution (an arbitrarily high number compared to the range of fitness values that a legal solution can have).

Genetic Operators. At the end of each generation, the reproduction operator is used on 10% of the individuals, and the crossover operator is used on 90% of

Table 1. Initialisation parameters of each genetic programming run

Population size	1000
Maximum depth of initial trees	4
Method of initialising the trees	Grow
Maximum generations	50

the individuals. These are standard parameters taken from [3]. In the crossover operator, any node in the tree can be selected with equal probability to be the crossover point. 'Fitness proportional selection' is used [3] (also known as 'roulette wheel selection'). Each individual is selected for the genetic operators with probability proportional to its normalised fitness. Reselection is permitted, so the original individuals involved in the genetic operations are put back in the population and can be used if selected again.

Function and Terminal Set. In the diagrams below, the functions and terminals are represented by symbols. The symbols are shown in table 2 along with an explanation of their function within the program. 'S', 'F' and 'C' are parameters given to the individuals in the population before they are evaluated.

Table 2. The functions and terminals used in the experiments and descriptions of the values they return

Symbol	Arguments	Description
+	2	Add
-	2	Subtract
*	2	Multiply
%	2	Protected divide function. Division by zero will return 1
<	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
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

4 Results

4.1 Evolved Small Trees

Figs. 2-5 show four *best-of-run* individuals (referred to as trees A-D) from four different runs, which illustrate that very 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 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.

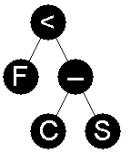

Fig. 2. Tree A

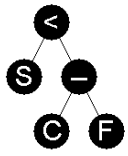

Fig. 3. Tree B

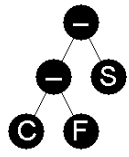

Fig. 4. Tree C

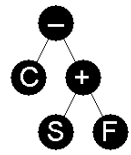

Fig. 5. Tree D

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. Therefore, they 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, which means the greater-than-zero condition, shown in the pseudo code (Fig. 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.

The question can be asked why trees C and D were the *best-of-run* individuals in their run when it seems easy for the system to produce trees which copy the functionality of trees A and B. The answer to this question is that if a tree like C or D is found first in its run then it is stored as the *best-of-run* individual so far. Trees like A and B *were* produced in the run but with the data sets used here they did not produce a solution which used less bins, so tree C or D remained as the best individual in the run so far.

The reason a better result is not gained by using trees 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, so 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.

This plateau in the search space also means that code-bloat [26] was not an issue in the runs which produced Figs. 2-5. 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. Code-bloat does exist in our genetic programming system, but the average and maximum complexity of the trees only increases in later generations. Therefore, if a first-fit 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.

4.2 More Complex Trees

In this section, we present two more examples of trees created during different runs which are more complex than the four individuals presented in section 4.1. These individuals are of interest because they are quite far removed from any

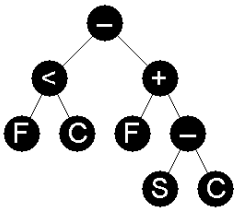

Fig. 6. Tree Structure 1

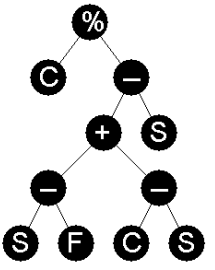

Fig. 7. Tree Structure 2

program that a human programmer would create for the same task. Both are *best-of-run* individuals.

Tree Structure 1. Fig. 6 has the same functionality as the first-fit heuristic. The tree works this way because F is always less than or equal to C. So 1 will always be returned by the left branch. In general, if the following equation is satisfied, then this means that the piece fits in the bin,

$$(F + (S - C)) <= 0$$

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.

Tree Structure 2. Fig. 7 shows a tree which also implements the first-fit heuristic. In Fig. 7, the tree's right hand branch (from the first subtract node downwards) can be simplified to:

$$(C - F) - S$$

This branch will therefore return 0 or greater if the piece fits or else it will return a negative number. 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 a negative number is returned by the right hand branch then the piece is not placed in the bin.

The most interesting feature of Fig. 7, however, is the way that the 'protected divide' node at the top fixes the problem that Figs. 4 and 5 have. Recall that in those trees, when the piece fits exactly, 0 is returned, and therefore the 'greater-than-zero' condition of Fig. 1 is not satisfied and the piece is not placed in the bin. The protected divide function at the root of Fig. 7 means that when the piece fits exactly, 1 is returned as it is a division by zero. So the problems in Figs. 4 and 5 have been solved in Fig. 7.

This functionality was not the reason why the protected divide function was defined in this way. The reason was to represent the closure of the function set,

as explained in [3]. 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.

4.3 Comparison with First-Fit and Other Heuristics Found

Over the 20 benchmark binpacking instances, our best evolved heuristic matches the performance of the first-fit heuristic (a well known human designed heuristic). This paper concentrates on reporting the best of the heuristics found. However, a worse heuristic was found in approximately 3 percent of the runs. For example, a heuristic with 17 nodes was found which puts a piece in a bin if its size is greater than the fullness multiplied by 2. Another heuristic with 17 nodes was found which never fills a bin higher than 148.

5 Conclusions and Future Work

It has been shown that a heuristic invented by a human, namely the first fit heuristic, can also be evolved by genetic programming. We evolved the heuristic without any human input except for the specification of the building-block nodes that it manipulates to make the tree structures of each individual in the population.

The selection pressure exerted by the fitness-proportional selection progressively eliminated those individuals that produced illegal solutions and used more individuals that produced legal solutions to construct the next generation. In this way the average fitness of the population improved over successive generations and better individuals were more likely to be created by the crossover genetic operator.

One potential area of future work is to increase the range of functions and terminals available to the genetic programming system. This will concentrate on expanding the potential complexity of evolved programs. Specifically, work will concentrate on expanding two new aspects of the system. Firstly, each individual must be given the array of bins as input, not just the piece size, bin fullness, and bin capacity. Secondly, automatically defined loops and automatically defined storage have to be included, which are explained in [2].

Plateaus in the search space are common when using the fitness function in this paper. We will investigate the use of other fitness functions which can differentiate between two solutions where the number of bins used is the same. An example is given in [27], where, if the number of bins used in both situations is equal, a solution with some full bins and some almost empty bins is considered superior to a solution where all bins are packed to a constant level.

We will test the genetic programming system over more binpacking benchmark instances to investigate if different heuristics are evolved under different conditions.

We will also investigate the effects of code-bloat as the population size is reduced from 1000. The average complexity of the population increases with the generation number, and we predict that as the population size is decreased,

the *best-of-run* individuals will be larger and harder to understand because the system will need more generations to find the same standard of program.

References

1. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge, Massachusetts (1994)
2. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem solving. Morgan Kaufmann (1999)
3. Koza, J.R.: Genetic Programming: on the Programming of Computers by Means of Natural Selection. The MIT Press, Boston, Massachusetts (1992)
4. Ross, P.: Hyper-heuristics. In Burke, E.K., Kendall, G., eds.: Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques. Kluwer, Boston (2005) 529–556
5. Burke, E.K., Hart, E., Kendall, G., Newall, J., Ross, P., Schulenburg, S.: Hyper-heuristics: An emerging direction in modern search technology. In Glover, F., Kochenberger, G., eds.: Handbook of Meta-Heuristics. Kluwer (2003) 457–474
6. Soubeiga, E.: Development and Application of Hyperheuristics to Personnel Scheduling. PhD thesis, Univesity of Nottingham, School of Computer Science (2003)
7. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **4** (1997) 67–82
8. Whitley, D., Watson, J.P.: Complexity theory and the no free lunch theorem. In Burke, E.K., Kendall, G., eds.: Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques. Kluwer, Boston (2005) 317–339
9. Ross, P., Schulenburg, S., Marin-Blazquez, J.G., Hart, E.: Hyper heuristics: Learning to combine simple heuristics in bin packing problems. In: Proceedings of the Genetic and Evolutionary Computation Conference 2002 (GECCO '02). (2002) 942–948
10. Ross, P., Marin-Blazquez, J.G., Schulenburg, S., Hart, E.: Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyper-heurstics. In: Proceedings of the Genetic and Evolutionary Computation Conference 2003 (GECCO '03), Chicago, Illinois (2003) 1295–1306
11. Burke, E.K., Kendall, G., Landa Silva, J.D., O'Brien, R.F.J., Soubeiga, E.: An ant algorithm hyperheuristic for the project presentation scheduling problem. In: Proceedings of the Congress on Evolutionary Computation 2005 (CEC'05). Volume 3., Edinburgh, U.K. (2005) 2263–2270
12. Burke, E.K., Kendall, G., Soubeiga, E.: A tabu-search hyper-heuristic for timetabling and rostering. Journal of Heuristics **9** (2003) 451–470
13. Cowling, P., Kendall, G., Soubeiga, E.: A hyperheuristic approach to scheduling a sales summit. In Burke, E.K., Erben, W., eds.: Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling (PATAT 2000), Konstanz, Germany (2000) 176–190
14. Burke, E.K., Landa Silva, J.D., Soubeiga, E.: Multi-objective hyper-heuristic approaches for space allocation and timetabling. In Ibaraki, T., Nonobe, K., Yagiura, M., eds.: Meta-heuristics: Progress as Real Problem Solvers, Selected Papers from the 5th Metaheuristics International Conference (MIC 2003). Springer (2005) 129–158

15. Burke, E.K., McCollum, B., Meisels, A., Petrovic, S., Qu, R.: A graph-based hyper heuristic for educational timetabling problems. European Journal of Operational Research (In Press, to appear 2006, Available online 21 November 2005)
16. Burke, E.K., Petrovic, S., Qu, R.: Case based heuristic selection for timetabling problems. Journal of Scheduling **9** (2006) 99–113
17. Dowsland, K., Soubeiga, E., Burke, E.K.: A simulated annealing hyper-heuristic for determining shipper sizes. Accepted for European Journal of Operational Research (In Press, to appear 2006, Available online 29 November 2005)
18. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley and Sons, Chichester (1990)
19. Falkenauer, E.: A hybrid grouping genetic algorithm for bin packing. Journal of Heuristics **2** (1996) 5–30
20. Beasley, J.E.: Binpacking benchmark data, at the brunell university or-library. Available at: http://people.brunel.ac.uk/∼mastjjb/jeb/orlib/binpackinfo.html (Last modified: 07-09-2004) [Accessed 1st March 2006].
21. Coffman Jr, E.G., Galambos, G., Martello, S., Vigo, D.: Bin packing approximation algorithms: Combinatorial analysis. In Du, D.Z., Pardalos, P.M., eds.: Handbook of Combinatorial Optimization. Kluwer (1998)
22. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, San Fransisco (1979)
23. Rhee, W.T., Talagrand, M.: On line bin packing with items of random size. Math. Oper. Res. **18** (1993) 438–445
24. Johnson, D., Demers, A., Ullman, J., Garey, M., Graham, R.: Worst-case performance bounds for simple one-dimensional packaging algorithms. SIAM Journal on Computing **3** (December 1974) 299–325
25. Koza, J.R., Poli, R.: Genetic programming. In Burke, E.K., Kendall, G., eds.: Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques. Kluwer, Boston (2005) 127–164
26. Bernstein, Y., Li, X., Ciesielski, V., Song, A.: Multiobjective parsimony enforcement for superior generalisation performance. In: Proceedings of the Congress for Evolutionary Computation 2004 (CEC'04), Portland, Oregon (2004) 83–89
27. Falkenauer, E., Delchambre, A.: A genetic algorithm for bin packing and line balancing. In: Proceedings of the IEEE 1992 Int. Conference on Robotics and Automation, Nice, France (1992) 1186–1192