Automating the Packing Heuristic Design Process with Genetic Programming

Edmund K. Burke ekb@cs.nott.ac.uk
University of Nottingham, School of Computer Science, Nottingham, UK

Matthew R. Hyde mvh@cs.nott.ac.uk
University of Nottingham, School of Computer Science, Nottingham, UK

Graham Kendall gxk@cs.nott.ac.uk
University of Nottingham, School of Computer Science, Nottingham, UK

John Woodward jrw@cs.nott.ac.uk
University of Nottingham, School of Computer Science, Nottingham, UK

Abstract
The literature shows that one, two and three dimensional bin packing and knapsack packing are difficult problems in operational research. Many techniques, including exact, heuristic, and metaheuristic approaches, have been investigated to solve these problems and it is often not clear which method to use when presented with a new instance. This paper presents an approach which is motivated by the goal of building computer systems which can design heuristic methods. The overall aim is to explore the possibilities for automating the heuristic design process.

We present a genetic programming system to automatically generate a good quality heuristic for each instance. It is not necessary to change the methodology depending on the problem type (one, two or three dimensional knapsack and bin packing problems), and it therefore has a level of generality unmatched by other systems in the literature. We carry out an extensive suite of experiments and compare with the best human designed heuristics in the literature. Note that our heuristic design methodology uses the same parameters for all the experiments.

The contribution of this paper is to present a more general packing methodology than those currently available, and to show that, by using this methodology, it is possible for a computer system to design heuristics which are competitive with the human designed heuristics from the literature. This represents the first packing algorithm in the literature able to claim human competitive results in such a wide variety of packing domains.

Keywords
genetic programming, genetic algorithms, evolutionary design, cutting and packing, hyper-heuristics

1 Introduction

In this paper we address the bin packing and knapsack problems, both of which have been intensely studied in the literature. This introduction will summarise the motivation for this work, and provide a literature review.

1.1 The Focus of this Paper

Our goal is to present a general methodology for packing problems, where any packing problem can be addressed by using the same system for all problem instances. We investigate a genetic programming methodology to evolve constructive heuristics for a collection of 18 benchmark knapsack and bin packing data sets, which are described in section 5. The collection consists of one, two and three dimensional problems. Many different heuristic and metaheuristic methods have been used previously, but no single methodology has been able to be applied to *all* of these data sets.

When presented with a new packing instance, a practitioner can select a methodology from the literature, or create a new bespoke packing heuristic for the instance. Both of these options take time and effort. The alternative we present here is that our more general methodology could be applied, to *automatically* create a new heuristic for the instance. We show that a more general methodology is possible, and that it is not necessary to sacrifice the quality of the results in order to acheive such generality.

Our methodology evolves a constructive heuristic, which decides which piece to place next and where it should be placed. These two decisions are made at the same timestep by the heuristic. This is in contrast to other heuristic approaches, where the order of the pieces to be packed is fixed before the packing starts. It is often the case that the pieces are pre-ordered by size from largest to smallest, as it is generally assumed that larger items are harder to pack and should be allocated space first. However, it is not currently possible to say with certainty which ordering will produce the best result for a given instance. Using the methodology described in this paper, the performance of an evolved heuristic is independent of any piece order.

Metaheuristic approaches have been used to generate the order of the pieces to be packed, before using a constructive heuristic to pack the fixed order. Usually a heuristic is used that has performed well in previous work. For example, it is shown in Hopper and Turton (2001) that the 'bottom-left-fill' heuristic performs better on average for the 2D strip packing problem when combined with a genetic algorithm and simulated annealing in this way. However, we cannot say which constructive heuristic will be superior on a given instance, and the heuristic choice is left to a human designer. Our approach presented in this paper avoids these limitations, by automating the design of the constructive packing method.

1.2 Hyper-Heuristics

One of the key goals of hyper-heuristic research is to "raise the level of generality at which optimisation systems can operate" (Burke et al., 2003a). A hyper-heuristic is defined as a heuristic which searches a space of heuristics, as opposed to a space of problem solutions, as explained in Ross (2005). There are (at least) two classes of hyper-heuristic (Burke et al., 2010), explained in the two sub-sections below. One class aims to intelligently choose heuristics from a predefined set. The other class aims to automatically generate heuristics from a set of components. It is this second class that is the focus of this paper.

1.2.1 Hyper-Heuristics to Choose Heuristics

In the majority of previous work, the hyper-heuristic is given a set of human created heuristics. These are often heuristics taken from the literature, that have been shown to perform well. On a given problem instance, the performance of the heuristics varies when they are applied individually, and therefore it is difficult to decide which heuristic to use for a given situation. When employing this type of hyper-heuristic approach, the hyper-heuristic is used to choose which heuristic to apply depending on the current problem state. The strengths of the heuristics can potentially be combined, and the decision of which heuristic to use is taken away from the user. This raises the generality of the system, because, as a whole, it can potentially be applied to many different instances of a problem, or to different problems, and maintain its performance.

Many metaheuristics and machine learning techniques have been used as hyper-heuristics. For example, Dowsland et al. (2007) use a simulated annealing hyper-heuristic for the shipper rationalisation problem. Ross et al. (2003) use a genetic algorithm as a hyper-heuristic. Case based reasoning is investigated as a hyper-heuristic by Burke et al. (2006c). Burke et al. (2003b) present a tabu search hyper-heuristic to two different problem domains, obtaining good results on both. Cuesta-Canada et al. (2005) use an ant algorithm to evolve sequences of (at most) five heuristics for 2D packing.

1.2.2 Hyper-Heuristics to Create Heuristics

In this paper, we use genetic programming as a hyper-heuristic, to evolve a *new* heuristic for a given problem instance. This is in contrast to the majority of previous work where the heuristics are provided manually to the algorithm. We define a number of functions and terminals that can be used as components to construct the heuristic. The genetic programming is a hyper-heuristic in this case because it searches a space of *all* the heuristics that can be created from the functions and terminals, rather than a static fixed set of pre-defined heuristics. A survey of this class of hyper-heuristic is presented in Burke et al. (2009b)

Previous work using a hyper-heuristic to create new heuristics has been by Fukunaga (2002, 2004, 2008) on the SAT problem, and Keller and Poli (2007) on the travelling salesman problem. There has also been work by Geiger et al. (2006) on the job shop problem, using genetic programming to evolve dispatching rules. Genetic programming has also been used as a hyper-heuristic for the one dimensional bin packing problem (Burke et al., 2006b, 2007a,b), evolving heuristics which are re-usable on new problem instances.

Terashima-Marin et al. (2005, 2006, 2007, 2008) use a genetic algorithm to evolve hyper-heuristics for the 2D packing problem domain. After each piece has been packed, the evolved hyper-heuristic decides which packing heuristic to apply next, based on the properties of the pieces left to pack. The genetic algorithm evolves a mapping from these properties to an appropriate heuristic. This sequence of papers follows work evolving similar hyper-heuristics for the one dimensional bin packing problem by Ross et al. (2002, 2003).

1.3 One Dimensional Packing

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. A set of orders for different lengths must be fulfilled by cutting stock lengths into smaller pieces while minimising the wasted material.

An online bin packing problem is one where the pieces must be packed one at a time, and cannot be moved once they are allocated a place. Simple constructive heuristics for the online one dimensional bin packing problem are given by Rhee and Talagrand (1993); Coffman Jr et al. (1998); Johnson et al. (1974). Kenyon (1996) explains the best-fit heuristic, which is perhaps the best known heuristic for this problem. Best-fit constructs a solution by putting each piece in turn into the bin which has the least space remaining. The process is repeated until all pieces have been allocated to a bin.

The offline bin packing problem occurs when all of the pieces are known before the packing starts. Heuristics for this problem are often obtained by combining an online heuristic with a sorting algorithm, which arranges the pieces from highest to lowest before the packing begins. For example, 'first-fit-decreasing' is the first-fit heurisic applied to a problem after the pieces have been sorted Yue (1991). First-fit-decreasing is therefore an offline heuristic because the pieces must be made known for the sorting to occur.

Two algorithms are presented by Yao (1980). 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 is presented by Coffman Jr et al. (2000); Seiden et al. (2003); Richey (1991)

Evolutionary algorithms have also been applied to the one dimensional bin packing problem Falkenauer and Delchambre (1992); O'Neill et al. (2004). In these cases, the evolutionary algorithm operates directly on a space of candidate solutions. This is in contrast to the hyper-heuristic approach used in this paper, where the evolutionary algorithm operates on a space of heuristics.

1.4 Two Dimensional Packing

The two dimensional packing problem occurs in the real world when shapes are cut from one or more stock sheets of material, such as metal, glass, textiles or wood. The aim in this case is to find the minimum number of stock sheets that are required to obtain all of the shapes. Real world examples of two dimensional cutting problems are investigated by Schneider (1988); Vasko et al. (1989); Lagus et al. (1996). This paper is concerned with packing orthogonal shapes. However, the literature also contains examples of problems with irregular shapes.

A common constraint placed on the cuts is the guillotine constraint. This means that each cut must be from one side of a piece to the other. Once the cut is made, the resulting two pieces of material are then free to be cut in the same way. In this paper the instances we use do not have this constraint, and with the exceptions explained in section 6.1.2, we allow 90° rotations of the pieces as they are packed or cut. In all cases we compare only to results in the literature which are obtained with the same constraints imposed.

A linear programming method is presented by Gilmore and Gomory (1961), but the results were obtained on small instances only. Tree search procedures have been employed more recently to produce optimal solutions for the 2D guillotine stock cutting problem Christofides and Whitlock (1977) and 2D non-guillotine stock cutting problem Beasley (1985). Also, Martello and Vigo (1998) use a branch and bound algorithm for the exact solution to the problem. More recent exact methodologies are presented by Clautiaux et al. (2008); Kenmochi et al. (2009); Macedo et al. (2010), and Alvarez-Valdes et al. (2009).

Bengtsson (1982) presents an early heuristic approach to the two dimensional bin packing problem. 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. The 'bottom left' heuristic (Baker et al., 1980) constructs a solution by sliding pieces repeatedly down and to the left.

Recently, the 'best-fit' heuristic was presented by Burke et al. (2004). This does not need to pre-order the pieces as 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 (Burke et al., 2004). This heuristic has been hybridised with simulated annealing to obtain even better results (Burke et al., 2009a).

Other metaheuristic methodologies have also been employed for two dimensional packing. For the two dimensional knapsack problem, Egeblad and Pisinger (2009) present a heuristic approach, using a local search on a sequence pair representation, with a simulated annealing heuristic providing the acceptance criteria. They test these heuristics both on the classical instances and on new instances which they define. A genetic algorithm for a two dimensional knapsack problem is presented by Hadjiconstantinou and Iori (2007), and different versions of genetic algorithms are also employed by Hwang et al. (1994), for three types of two dimensional packing problems. Burke et al. (2006a) present a bottom-left-fill heuristic algorithm which employs tabu search, for the two dimensional packing problem with irregular shapes.

1.5 Three Dimensional Packing

The packing of goods into standard sized containers is common in manufacturing and transportation. The goal is to maximise the volume utilisation of the containers, or to minimise the number of containers that are needed to hold all of the goods. The containers would then be loaded onto a vehicle for transport.

For the three dimensional bin packing problem, Ivancic et al. (1989) present an exact method using an integer programming representation. Bischoff and Ratcliff (1995) address some differences between the real world problems and the less constrained problem instances from the literature. They describe two algorithms, one to create stable packings, and one to create convenient packings for if the pieces are to be unloaded at more than one stop. Eley (2002) developed 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. Lim and Zhang (2005) present a successful iterative approach for both the bin packing and the knapsack problem. This approach uses a greedy heuristic and tree search. 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.

The three dimensional knapsack problem has also received significant interest in the literature. Ngoi et al. (1994) 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. Chua et al. (1998) also use a similar spacial representation technique. The hybrid genetic algorithm of Bortfeldt and Gehring (2001) 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. Lim et al. (2003) present a multi-faced buildup method. The representation for this allows every surface in the container to be a 'floor' to locate boxes. Egeblad and Pisinger (2009) use a heuristic approach using 'sequence triples', which is based on the sequence pair representation mentioned in section 1.4. Huang and He (2009) present a packing algorithm using a concept called 'caving degree' that they define. 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.

2 Problem Description

2.1 The Knapsack Problem

The one dimensional 0-1 knapsack problem consists of a set of pieces j, each with a weight w_j and a value v_j. The pieces must be packed into one knapsack with capacity c.

Not all of the pieces can fit into the knapsack, so the objective is to maximise the value of the pieces chosen to be packed. A mathematical formulation of the 0-1 knapsack problem is shown in equation 1, taken from Martello and Toth (1990), where x_j is a binary variable indicating whether piece j is selected to be packed into the knapsack.

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

The knapsack problem can be defined in two (and three) dimensions. The knapsack has a width W and 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), and a value v_j. 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 by Christofides and Whitlock (1977).

2.2 The Bin Packing Problem

The classical one dimensional bin packing problem is similar to the knapsack problem. The difference is that *all* of the pieces must be packed, and an unlimited number of bins are available. The objective is to minimise the number of bins necessary to accommdate all of the pieces. A mathematical formulation of the bin packing problem is shown in equation 2, taken from Martello and Toth (1990). Where n is the number of pieces (and therefore also the maximum amount of bins necessary), 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.

$$\begin{aligned}
&\text{Minimise} && \sum_{i=1}^{n} y_i && \\
&\text{Subject to} && \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, && i \in N = \{1,\ldots,n\}, \\
& && \sum_{i=1}^{n} x_{ij} = 1, && j \in N, \\
& && y_i \in \{0,1\}, && i \in N, \\
& && x_{ij} \in \{0,1\}, && i \in N, j \in N, \qquad (2)
\end{aligned}$$

The bin packing problem can be defined in two (and three) dimensions in the same way as for the knapsack problem. The objective is to minimise the number of bins needed to accommodate all of the items. In this paper we allow rotations of the pieces in all directions, except where the instance itself specifies that only certain rotations are allowed for each piece.

3 Representation for One, Two, and Three Dimensional Packing Problems

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 3.5 to

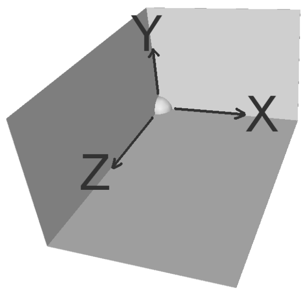

Figure 1: An initialised bin with one 'corner' in the back-left-bottom corner of the bin

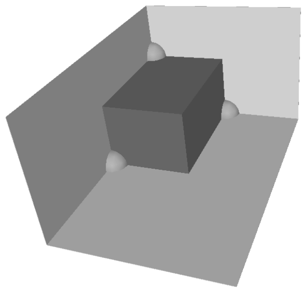

Figure 2: A bin with one piece placed in the corner from figure 1

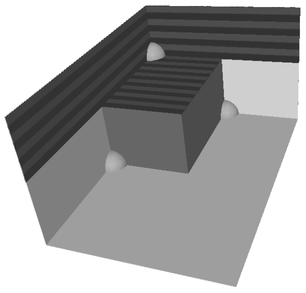

Figure 3: The three surfaces defined by the corner that is in the Y direction of the piece

cover the bin packing problem and to cover problems of lower dimensions.

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 1. Therefore at the start of the packing, the heuristic just chooses which piece to put into this corner, because it is the only one available. Figure 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 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 intersect to define it, in the XY plane, the XZ plane, and the YZ plane. The corner is at the intersection of these three orthogonal planes, and the size and limits of the three surfaces are defined by the extent of the piece faces (or container faces) that intersect at the corner. Figure 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, while the XY and YZ surfaces are limited by the edges of the container. Similarly, figure 4 shows the three surfaces of the corner that is to the right of the piece in the figure. Each surface has a length along each of the two axes of the plane to which it belongs. So an XZ surface will have a length in the X direction and a length in the Z direction.

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 at that corner. Figure 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 6, the new piece exceeds the limit of the corner's YZ surface in the Z direction, so this placement is invalid.

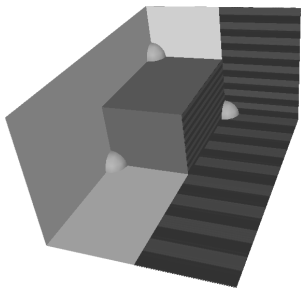
Figure 4: The three surfaces defined by the corner that is in the X direction of the piece

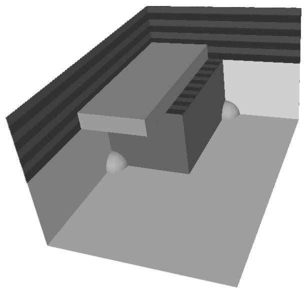
Figure 5: An invalid placement of a new piece, because it exceeds the limit of the corner's XZ surface in the Z direction

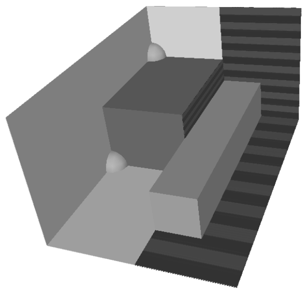
Figure 6: An invalid placement of a new piece, because it exceeds the limit of the corner's YZ surface in the Z direction

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, where the piece 'P' is placed in the middle of three other existing pieces, 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 now has two surfaces in its XZ plane, and the corner on the right has two surfaces in its YZ plane.

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, as no piece can be put there at any time in the packing process. If left unchecked, eventually there will be many corners of this type that no piece can fit into, and there may potentially be 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 in 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 limits of the corner's surfaces that are closest to the corner itself. So the filler pieces also do not exceed any of the limits of the three surfaces of a corner. In figure 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 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 9, three corners have their surfaces updated by the filler piece. First, the corner in the

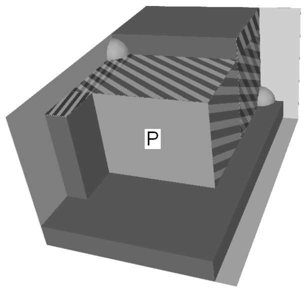

Figure 7: A piece which reaches the limit of two surfaces of the corner that it was put into

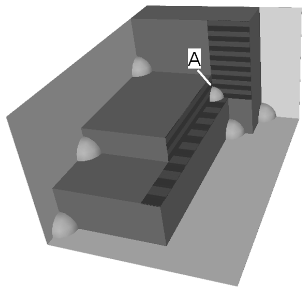

Figure 8: The three surfaces of the corner with the smallest avaliable area

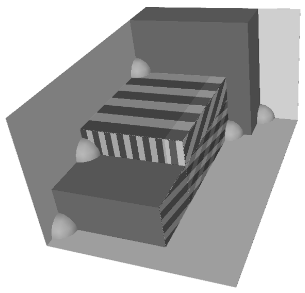

Figure 9: A filler piece is put into corner 'A' from figure 8, the surfaces of two corners are updated

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. As soon as the smallest corner can accept at least one piece from those remaining, we continue the packing. In the knapsack problem there is only one bin. The packing will terminate when the whole bin is filled with pieces and filler pieces, so there are no corners left. The result is then the total value of the pieces in the knapsack.

3.5 Bin Packing, and Packing in Lower Dimensions

The representation described in sections 3.1-3.4 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 in it, if this bin is used then a new one 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 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.

4 The Genetic Programming Methodology

This section describes the evolutionary algorithm that evolves packing heuristics. Each program in the genetic programming population is a packing heuristic, and is applied to a packing problem to obtain a result. The heuristics are then manipulated by the genetic programming system depending on their performance. 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'. Sections 4.1 and 4.2 explain how a heuristic is *applied*, and sections 4.3 and 4.4 explain how the population of heuristics are *evolved*.

4.1 How the Heuristic is Applied

The heuristic (an individual in the genetic programming population) operates within a fixed framework which, at each decision point, evaluates the heuristic once for each valid allocation that can be performed. The individuals are tree structures, with internal (function) and leaf (terminal) nodes. The terminal nodes represent variables which change their value depending on the problem state. To evaluate an individual, its terminal nodes are set to the values they represent, and the function nodes perform operations on the values returned by their child nodes. More information on functions and terminals can be found in introductory genetic programming references (Koza, 1992; Banzhaf et al., 1998; Koza and Poli, 2005)

The heuristic returns one numerical value each time it is evaluated, which is then interpreted as a score for the allocation. So every piece, orientation and corner combination, in every bin, is evaluated in this way. The actual allocation performed is the one which receieves the maximum evaluation (score). Then the filler stage (section 3.4) is performed to fill any redundant space, and the cycle then repeats at the next decision point. This process is shown in algorithm 1.

To evaluate a heuristic for a valid allocation, its terminal values are set to values dictated by the properties of the current allocation, i.e. the properties of the current piece, orientation and corner (see section 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 suitabilility of the allocation.

When all of the possible allocations have been considered by the heuristic, then the one for which the heuristic returned the highest value is performed, by putting the piece into the corner in its chosen orientation. The corner structure is then updated because the new piece will create new corners.

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. When this occurs, the knapsack packing is complete, and the heuristic has been used to form a solution. In summary, the heuristic chooses 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.

4.2 Genetic Programming Functions and Terminals

Table 1 summarises the twelve functions and terminals and the values they return, the functions are shown in the top four rows, and the terminals in the lower eight rows. The arithmetic operators add, subtract, multiply and protected divide are chosen to be included in the function set. Genetic programming usually uses 'protected divide' instead of the standard divide function, because there is always a possibility that the denominator will be zero. The protected divide function replaces a zero denominator by 0.001.

The first two terminals shown in table 1 represent attributes of the piece. The first

Algorithm 1 Pseudo code showing the overall program structure within which a heuristic operates. The heuristic assigns a score to each potential allocation, and the best is actually performed.

```
while pieces exist to be packed do
  if no piece can fit into the corner with the smallest area then
    put a filler piece into this corner and update the corner structure
  else
    for all pieces P in pieceList do
      for all orientations O of the piece do
        for all corners C in current partial solution do
          if piece P in orientation O fits into corner C then
            currentAllocation = P,C,O
            x = evaluateHeuristic(currentAllocation)
            if x > bestx then
              bestAllocation = currentAllocation
              bestx = x
            end if
          end if
        end for
      end for
    end for
    perform bestAllocation on solution
    remove chosen piece from pieceList
    update corner structure
    if necessary, open a new bin containing one available corner
    if there are no corners left then
      knapsack packing is complete, break from while loop
    end if
  end if
end while
```

terminal represents the volume of the piece. If needed, some kind of piece priority based on size can be evolved because of the information this terminal provides to the heuristic. The second terminal represents the value of the piece. This information is useful only in the knapsack problem, but is still left in the terminal set when a heuristic is being evolved for bin packing, because one of our aims is to demonstrate that the proposed algorithm requires no parameter modification or tuning. Therefore, in the bin packing case, the value terminal will always return one. Both the volume and the value terminals could, for example, be used in the heuristic to prioritise pieces based on their value per unit volume.

The three ‘waste’ terminals give the heuristic information on how good a fit the piece is to the corner under consideration. There is one for each surface of 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 is 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.

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

Table 1: The functions and terminals and descriptions of the values they return

Name	Description
+	Add two inputs
-	Subtract second input from first input
*	Multiply two inputs
%	Protected divide function, divides the first input by the second
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

the relevant coordinate of the corner. For example, cornerY returns the Y coordinate of the current corner.

4.3 Fitness of a Heuristic

Each heuristic in the population packs the given instance, and its fitness is assigned to it based on the quality of the packing. The quality of the packing is necessarily calculated differently for bin packing and knapsack instances.

4.3.1 Bin Packing Fitness

For bin packing problems, we use the fitness function shown in equation 3, based on that presented by Falkenauer and Delchambre (1992), where n = number of bins, m = number of pieces. v_j = volume of piece j. x_{ij} = 1 if piece j is in bin i and 0 otherwise, and C = bin volume (capacity). Only the bins that contain at least one piece are included in this fitness function.

$$Fitness = 1 - \left(\frac{\sum_{i=1}^{n} \left(\frac{\sum_{j=1}^{m} v_j x_{ij}}{C} \right)^2}{n} \right) \quad (3)$$

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 (Burke et al., 2006b). 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.

4.3.2 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 to be the fitness because the genetic programming implementation treats lower fitness values as better. This fitness function is shown

Table 2: Parameters of each genetic programming run

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

in equation 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.

$$Fitness = \frac{1}{\sum_{j=1}^{n} v_j x_j} \tag{4}$$

4.4 Genetic Programming Parameters

Table 2 shows the parameters used in the genetic programming runs. The mutation operator uses the 'grow' method explained by Koza (1992), with a minimum and maximum depth of five, and 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 we used for our experiments.

5 The Data Sets

We have tested our methodology on a comprehensive selection of 18 data sets from the literature, which are summarised in table 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 as knapsack instances can be used as bin packing instances by ignoring the value of each piece, and packing all the pieces in the instance into the fewest bins possible. Instances originally intended as bin packing problems do not specify a value for each piece, so to use them as knapsack instances it is usual for practitioners to set each piece's value equal to its volume. The instances are used in a large number of papers. In table 3 we give the references where the instances have been used to obtain the results we have compared against in this paper. They use the same set of constraints, meaning we can compare fairly with these results.

6 Results

We obtain results on a suite of 18 data sets, the details of which are shown in section 5. Each data set contains a number of instances. We compare our results for an instance against the result of the best heuristic in the literature for that instance. Tables 4 and 5 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 5, 6 and 7.

For every instance, we perform ten runs, each with a different random seed. We calculate the average over the ten runs for the instance, and name this value the 'InstanceAverage'. This is shown in equation 5, where r_i is the result of run i.

Table 3: A summary of the 18 data sets used in this paper

	Instance Name	Number of Instances	Used For		References
			Bin Packing	Knapsack	
1D	Uniform	80	√	×	Falkenauer (1996)
	Hard	10	√	×	Scholl et al. (1997) Schwerin and Wascher (1997)
2D	beng	10	√	×	Bengtsson (1982) Martello and Vigo (1998)
	Okp	5	×	√	Egeblad and Pisinger (2009)
	Wang	1	×	√	
	Ep30	20	×	√	
	Ep50	20	×	√	
	Ep100	20	×	√	
	Ep200	20	×	√	
	Ngcut	12	√	√	Egeblad and Pisinger (2009) Martello and Vigo (1998)
	Gcut	13	√	√	
	Cgcut	3	√	√	
3D	Ep3d20	20	×	√	Egeblad and Pisinger (2009)
	Ep3d40	20	×	√	
	Ep3d60	20	×	√	
	Thpack8	15	×	√	Ngoi et al. (1994) Bischoff and Ratcliff (1995) Chua et al. (1998) Bortfeldt and Gehring (2001)
	Thpack9	47	√	√	Huang and He (2009) Ivancic et al. (1989) Bischoff and Ratcliff (1995) Eley (2002) Lim and Zhang (2005)
	BandR	700	×	√	Egeblad and Pisinger (2009) Bortfeldt and Gehring (2001) Bortfeldt et al. (2003) Lim et al. (2003) Lim and Zhang (2005)

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 6, where z is the best result in the literature for the given instance.

Each instance in a set has an InstanceRatio, and the 'Ratio' is the average InstanceRatio of all the instances in the set. This is shown in equation 7, where i is the instance number, and m is the number of instances in the data set.

$$InstanceAverage = \frac{\sum_{i=1}^{10} r_i}{10} \tag{5}$$

$$InstanceRatio = \frac{InstanceAverage}{z} \tag{6}$$

$$Ratio = \frac{\sum_{i=1}^{m} InstanceRatio_i}{m} \tag{7}$$

Tables 4 and 5 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 consistent, we convert the Ratios to a percentage value which represents 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 4 and 5.

6.1 Bin Packing Results

The discussion in this section refers to the results reported in table 4.

6.1.1 One Dimensional

For the 'Uniform' data set, we compare to Falkenauer (1996). Our results are one bin worse than Falkenauer's results for three instances out of 80. We 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 Wascher (1997), who have solved the instances to optimality. We achieve the optimal result for all but one instance of the set of ten, where our best result uses one bin more than the optimal. For 7 of the instances, we find the optimal result in all ten runs.

6.1.2 Two Dimensional

For the 'Beng' data set, our results are compared to the exact methods of Bengtsson (1982) and Martello and Vigo (1998). For the 'Ngcut', 'Gcut', and 'Cgcut' sets, we compare our results to Martello and Vigo (1998). 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 (1998) do not find a result for the eighth gcut instance. We 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.

Table 4: Summary of bin packing results, and the percentage improvement over the best results in the literature

	Instance Name	Ratio	Standard Deviation	% Difference
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

Table 5: Summary of knapsack results, and the percentage improvement over the best results in the literature

	Instance Name	Ratio	Standard Deviation	% Difference
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

6.1.3 Three Dimensional

Thpack9 is the only 3D bin packing instance. 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.

6.2 Knapsack Results

The discussion in this section refers to the results reported in table 5.

6.2.1 Two Dimensional

We compare all our two dimensional knapsack results to the results of Egeblad and Pisinger (2009), who define the four new 'Ep*' instance sets 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.

Our results on the nine problem sets show that there is no real difference between the performance of our evolved heuristics and the performance of the heuristic method presented by Egeblad and Pisinger (2009). There are five sets with ratio just greater than one, and four just 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.

6.2.2 Three Dimensional

There are three subsets in the ep3d set. In the first set, with instances of 20 pieces, our results are better in every instance than the results of Egeblad and Pisinger (2009). In the second set our results are better in all but two instances, and in the third set all but four of our results are better.

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, our system could not reach the results of the CBUSE method of Bortfeldt and Gehring (2001). However our result for the 2nd instance is second only to CBUSE in the literature, and the result for the 6th instance is third best in the literature.

Over the 47 instances of Thpack9, we obtain an average space utilisation of 95.2 percent, which is compared to the 94.6 percent of Huang and He (2009). We get 100 percent space utilisation in 14 instances out of the 47. The individual results for the 47 instances of this set are not reported, so in this data set we could not use equations 5, 6 and 7. Therefore, in the Ratio column of table 5 for this data set, we report the ratio of our average space utilisation over the average space utilisation of Huang and He (2009). This also means that the standard deviation is not reported because we could not calculate InstanceRatio values for each instance of this set.

There are 5 papers that report results for the BandR data set. There are 700 instances, split into 7 subsets of 100. Therefore, unsuprisingly, 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 with m set to seven. 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 An Example Evolved Heuristic

In this section we give an example of an evolved heuristic, which was evolved on the 3D knapsack instance BR5-0. This instance is the first in the set of 100 in the BR5 class, each of which have 12 different sizes of piece in different quantities. Figure 10 shows the heuristic in prefix notation, and figures 11 and 12 show the result on the BR5-0 instance from two different angles.

It is difficult to determine the exact behaviour of this heuristic as it is a complex expression, but it is possible to gain some insights by investigating the numerical results that it returns when the values of its terminals are manually modified. For example,

```
(- (+ (+ (- Y X) (+ (% (% v XZ) XZ) (% (+ (% X (+ (% (+ (- (% V X) XZ) (% v Z)) XZ)
(% v XZ))) (% (% XZ XZ) XZ)) (+ (- Y (+ (% X (% (- (% V X) XZ) (+ Y Y))) (% (% v
XZ) XZ))) (+ (% (% v (+ (+ (+ (% X (% (% v XZ) XZ)) (% v XZ)) (+ (% (% (% v XZ)
XZ) XZ) (% (% v XZ) XZ))) (+ (% X Z) (% (% v XZ) XZ)))) XZ) (% (+ (% X (+ (% (+
(% X (% v XZ)) (% v XZ)) XZ) (+ (% (- (% V X) XZ) (+ Y Y)) (% X XZ)))) (% v (- YZ
X))) XZ)))))) (+ (% (% (+ V (% v XZ)) XZ) XZ) (+ (% X (% (- (% V X) XZ) (+ Y Y))) (%
(% v XZ) XZ)))) XZ)
```

Figure 10: An example heuristic, which was evolved on the instance BR5-0, displayed in prefix notation. The terminal names are abbreviated to save space. ‘X’ ‘Y’ and ‘Z’ are the corner coordinates, ‘V’ is the piece value, ‘v’ is the piece volume, and ‘XZ’ is the XZWaste terminal.

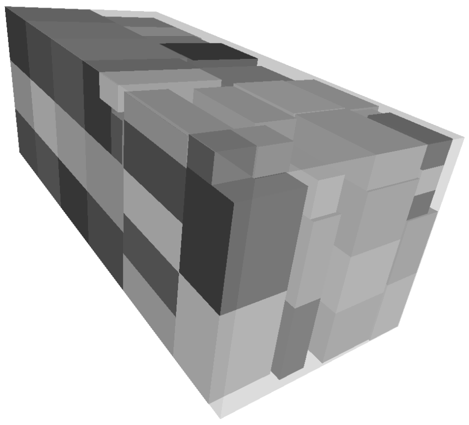

Figure 11: The solution to instance BR5-0 found by the heuristic in figure 10

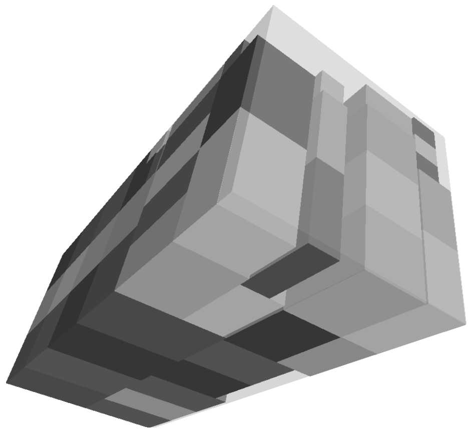

Figure 12: The same solution as figure 11, viewed from underneath

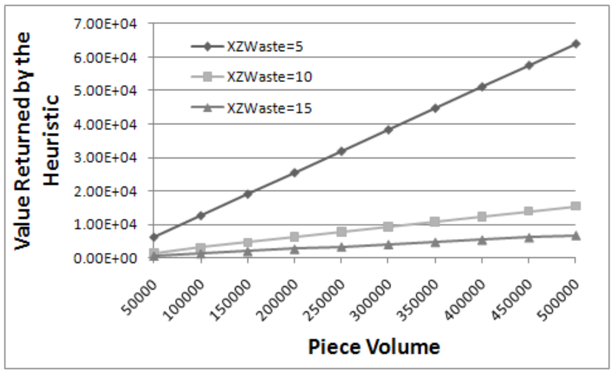

Figure 13: The values returned by the example evolved heuristic in figure 10, when the volume of the piece is changed for three different values of XZWaste. The graph shows that for any given volume of piece, it will be placed in the orientation that minimises its XZWaste.

when all of the other terminals are fixed, a piece with a higher volume will receive the highest result from the heuristic. This seems to suggest that when there is a choice of pieces to place into a corner, the largest piece will be chosen. However, the behaviour of the heuristic cannot be described as simply as this, because an increase in the volume of the box will usually result in a reduction in the waste on one or more of the three surfaces at the corner. Therefore, the other terminals cannot realistically be treated as fixed.

The only waste terminal that is included is 'XZWaste'. This means that wasted space on the XY and YZ surfaces are not considered by the heuristic, but the quality of the fit onto the piece below *does* form part of the decision process. Figure 13 shows the behaviour of the heuristic when the XZWaste is set to 5, 10 and 15, and the volume of the piece is increased from 50000 to 500000. All of the other terminals are fixed at 20. The graph shows that, in general, an allocation will be considered better if the XZ waste is lower, this is the dominating component especially at low values of XZWaste. However, another piece with more waste can be considered better if is at least a certain volume. For example, figure 13 shows that an allocation with XZWaste=5 and a piece volume of 100000 will be rated lower than an allocation with XZWaste=10 (more wasted space) and a piece volume of 350000 or greater. So there is a volume threshold, above which a piece with more XZWaste will be put in ahead of one that fits better onto the XZ surface. The graph also shows that a piece will be placed in the orientation that minimises its XZWaste.

The line for 'XZWaste=0' is not shown because it is significantly above the scale of the y-axis in figure 13. The values returned by the heuristic for a piece which has no XZWaste are of the order of 10^{14}, much higher than 10^{4} which is returned when there is an XZWaste of 5. If a piece fits exactly onto the XZ surface, then it will always be placed there regardless of the size of the pieces in alternative allocations.

The heuristic will mostly have a choice of more than one corner into which to put a piece. Figure 14 shows the effect of differing values for the CornerX, CornerY, and CornerZ terminals. This graph shows that the value returned by the heuristic decreases when the X coordinate of the corner increases, so the heuristic prefers to put pieces closer to the left side of the container. When the Y coordinate of the corner

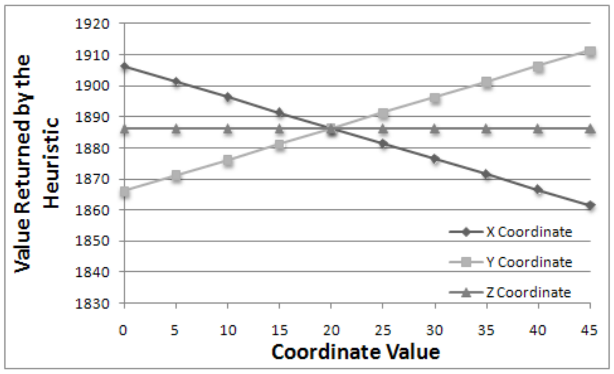

Figure 14: The values returned by the example evolved heuristic in figure 10, when the CornerX, CornerY, and CornerZ terminals change their values. The values are shown when each is set to between 0 and 45, with the other two terminals fixed at 20. The Volume and Value terminals are set to 200000, and the Waste terminals are fixed at 20. The graph shows that, with all other terminals being equal, a piece will be placed at a smaller X coordinate, and a larger Y coordinate. The Z coordinate makes very little difference.

increases, the value returned by the heuristic *increases*, so it prefers to put pieces into higher corners given the same X and Z coordinate. The Z coordinate has almost no effect on the value returned by the heuristic.

After considering all of these observations together, the behaviour of this example heuristic could be summarised as a 'tower building' approach. Allocations considered 'good' by the heuristic are those in which the piece fits well onto the surface below it, and are higher in the container. The distance from the back of the container is not considered important, but the pieces are generally allocated firstly to the left side of the container if there is a choice. This tower building behaviour can be seen to some extent in figures 11 and 12, where the left side of the container is filled with large identical pieces. The heuristic completes each tower of three pieces before starting the next one in front of it, showing that the heuristic considers the higher piece allocations to be better. More irregular towers can be seen at the front of the solution, where each piece fits well to the piece below.

8 Conclusions

In this paper, we have described a genetic programming methodology that evolves a heuristic for any one, two or three dimensional knapsack or bin packing problem. This methodology can be described as a hyper-heuristic, because it searches a space of heuristics rather than a space of solutions directly. It differs from a traditional hyper-heuristic methodology because its aim is not to choose from a set of pre-specified heuristics, but to automatically generate one heuristic from a set of potential components.

We have presented the results obtained by these automatically designed heuristics, and have shown them to be highly competitive with the state of the art human created heuristics and metaheuristics. Therefore, the contribution of this paper is to show that computer designed heuristics can at least equal the performance of human designed heuristics, in the packing problems addressed here. This is especially significant as these packing problems are well studied, and the human created heuristics obtain very close to optimal solutions for some of the data sets.

Automatic heuristic generation is a 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 large instances, and a variety of instance sets. Designing a metaheuristic system, and optimising its parameters, is a process that can take many more hours of research.

Currently, in the cutting and packing literature, systems are developed and optimised for one problem domain. For example, a metaheuristic system developed solely for three dimensional packing instances cannot operate on one dimensional instances. This paper represents the first packing system that can successfully operate over different problem domains. The system can be this general because it can generate new heuristics for each problem domain.

Therefore, in addition to showing that human competitive heuristics can be automatically designed, a further contribution of this paper is to present a system that can generate a solution for any one two or three dimensional knapsack or bin packing problem instance, with no change of parameters in between. All that is required is to provide the problem instance file, and specify whether it is a bin packing or knapsack instance. 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.

References

Alvarez-Valdes, R., Parreno, F., and Tamarit, J. M. (2009). A branch and bound algorithm for the strip packing problem. *OR Spectrum*, 31(2):431–459.

Baker, B. S., Coffman, E. G., and Rivest, R. L. (1980). Orthogonal packings in two dimensions. *SIAM J. Comput.*, 9(4):846–855.

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic programming, an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann, San Francisco.

Beasley, J. E. (1985). An exact two-dimensional non-guillotine cutting tree search procedure. *Oper. Res.*, 33(1):49–64.

Bengtsson, B. E. (1982). Packing rectangular pieces - a heuristic approach. *The Comput. J.*, 25(3):353–357.

Bischoff, E. and Ratcliff, M. (1995). Issues in the development of approaches to container loading. *Omega*, 23(4):377–390.

Bortfeldt, A. and Gehring, H. (2001). A hybrid genetic algorithm for the container loading problem. *Eur. J. of Oper. Res.*, 131(1):143–161.

Bortfeldt, A., Gehring, H., and Mack, D. (2003). A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*, 29(5):641–662.

Burke, E., Kendall, G., and Whitwell, G. (2004). A new placement heuristic for the orthogonal stock-cutting problem. *Oper. Res.*, 55(4):655–671.

Burke, E., Kendall, G., and Whitwell, G. (2009a). A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock cutting problem. *INFORMS J. On Computing*, 21(3):505–516.

Burke, E. K., Hart, E., Kendall, G., Newall, J., Ross, P., and Schulenburg, S. (2003a). Hyper-heuristics: An emerging direction in modern search technology. In Glover, F. and Kochenberger, G., editors, *Handbook of Meta-Heuristics*, pages 457–474. Kluwer, Boston, Massachusetts.

Burke, E. K., Hellier, R. S. R., Kendall, G., and Whitwell, G. (2006a). A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Oper. Res.*, 54(3):587–601.

Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Woodward, J. (2009b). Exploring hyper-heuristic methodologies with genetic programming. In Mumford, C. and Jain, L., editors, *Computational Intelligence: Collaboration, Fusion and Emergence*, pages 177–201. Springer.

Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J. (2010). A classification of hyper-heuristics approaches. In Gendreau, M. and Potvin, J., editors, *Handbook of Meta-Heuristics 2nd Edition*, pages 449–468. Springer.

Burke, E. K., Hyde, M. R., and Kendall, G. (2006b). Evolving bin packing heuristics with genetic programming. In Runarsson, T., Beyer, H.-G., Burke, E., J.Merelo-Guervos, J., Whitley, D., and Yao, X., editors, *LNCS 4193, Proc. of the 9th Internat. Conf. on Parallel Problem Solving from Nature (PPSN 2006)*, pages 860–869, Reykjavik, Iceland.

Burke, E. K., Hyde, M. R., Kendall, G., and Woodward, J. (2007a). Automatic heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one. In *Proc. of the 9th ACM Genetic and Evolutionary Computation Conf. (GECCO 2007)*, pages 1559–1565, London, UK.

Burke, E. K., Hyde, M. R., Kendall, G., and Woodward, J. (2007b). The scalability of evolved on line bin packing heuristics. In *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 2530–2537, Singapore.

Burke, E. K., Kendall, G., and Soubeiga, E. (2003b). A tabu-search hyper-heuristic for timetabling and rostering. *J. of Heuristics*, 9(6):451–470.

Burke, E. K., Petrovic, S., and Qu, R. (2006c). Case-based heuristic selection for timetabling problems. *J. of Scheduling*, 9(2):115–132.

Christofides, N. and Whitlock, C. (1977). An algorithm for two-dimensional cutting problems. *Oper. Res.*, 25(1):30–44.

Chua, C. K., Narayanan, V., and Loh, J. (1998). Constraint-based spatial representation technique for the container packing problem. *Integrated Manufacturing Systems*, 9(1):23–33.

Clautiaux, F., Jouglet, A., Carlier, J., and Moukrim, A. (2008). A new constraint programming approach for the orthogonal packing problem. *Computers and Operations Research*, 35(3):944–959.

Coffman Jr, E. G., Courcoubetis, C., Garey, M., Johnson, D. S., Shor, P. W., Weber, R. R., and Yannakakis, M. (2000). 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.

Coffman Jr, E. G., Galambos, G., Martello, S., and Vigo, D. (1998). Bin packing approximation algorithms: Combinatorial analysis. In Du, D. Z. and Pardalos, P. M., editors, *Handbook of Combinatorial Optimization*. Kluwer.

Cuesta-Canada, A., Garrido, L., and Terashima-Marin, H. (2005). Building hyper-heuristics through ant colony optimization for the 2d bin packing problem. In *LNCS 3684, Proc. of the 9th Internat. Conf. on Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, pages 654–660, Melbourne, Australia.

Dowsland, K., Soubeiga, E., and Burke, E. K. (2007). A simulated annealing hyper-heuristic for determining shipper sizes. *Eur. J. of Oper. Res.*, 179(3):759–774.

Egeblad, J. and Pisinger, D. (2009). Heuristic approaches for the two- and three-dimensional knapsack packing problem. *Comput. and Oper. Res.*, 36(4):1026–1049.

Eley, M. (2002). Solving container loading problems by block arrangement. *Eur. J. of Oper. Res.*, 141(2):393–409.

Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *J. of Heuristics*, 2:5–30.

Falkenauer, E. and Delchambre, A. (1992). A genetic algorithm for bin packing and line balancing. In *Proc. of the IEEE 1992 Int. Conf. on Robotics and Automation*, pages 1186–1192, Nice, France.

Fukunaga, A. S. (2002). Automated discovery of composite sat variable-selection heuristics. In *Eighteenth national Conf. on Artificial intelligence*, pages 641–648, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Fukunaga, A. S. (2004). Evolving local search heuristics for SAT using genetic programming. In Deb, K., Poli, R., Banzhaf, W., Beyer, H.-G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P. L., Spector, L., Tettamanzi, A., Thierens, D., and Tyrrell, A., editors, *LNCS 3103. Proc. of the ACM Genetic and Evolutionary Computation Conf. (GECCO '04)*, pages 483–494, Seattle, WA, USA. Springer-Verlag.

Fukunaga, A. S. (2008). Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation (MIT Press)*, 16(1):31–1.

Geiger, C. D., Uzsoy, R., and Aytug, H. (2006). Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *J. of Scheduling*, 9(1):7–34.

Gilmore, P. and Gomory, R. (1961). A linear programming approach to the cutting-stock problem. *Oper. Res.*, 9(6):849–859.

Hadjiconstantinou, E. and Iori, M. (2007). A hybrid genetic algorithm for the two-dimensional single large object placement problem. *Eur. J. of Oper. Res.*, 183(3):1150–1166.

Hopper, E. and Turton, B. C. H. (2001). An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *Eur. J. of Oper. Res.*, 128(1):34–57.

Huang, W. and He, K. (2009). A new heuristic algorithm for cuboids packing with no orientation constraints. *Computers and Operations Research*, 36(2):425–432.

Hwang, S. M., Kao, C. Y., and Horng, J. T. (1994). On solving rectangle bin packing problems using genetic algorithms. In *Proc. of the 1994 IEEE Internat. Conf. on Systems, Man, and Cybernetics*, volume 2, pages 1583–1590, IEEE, San Antonio, TX.

Ivancic, N., Mathur, K., and Mohanty, B. (1989). An integer-programming based heuristic approach to the three-dimensional packing problem. *J. of Manufacturing and Oper. Management*, 2:268–298.

Johnson, D., Demers, A., Ullman, J., Garey, M., and Graham, R. (1974). Worst-case performance bounds for simple one-dimensional packaging algorithms. *SIAM J. on Computing*, 3(4):299–325.

Keller, R. E. and Poli, R. (2007). Linear genetic programming of parsimonious metaheuristics. In *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 4508–4515, Singapore.

Kenmochi, M., Imamichi, T., Nonobe, K., Yagiura, M., and Nagamochi, H. (2009). Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198(1):73–83.

Kenyon, C. (1996). Best-fit bin-packing with random order. In *Proc. of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 359–364.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. The MIT Press, Boston, Massachusetts.

Koza, J. R. and Poli, R. (2005). Genetic programming. In Burke, E. K. and Kendall, G., editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 127–164. Kluwer, Boston.

Lagus, K., Karanta, I., and Yla-Jaaski, J. (1996). Paginating the generalized newspaper, a comparison of simulated annealing and a heuristic method. In *Proc. of the 5th Internat. Conf. on Parallel Problem Solving from Nature (PPSN '96)*, pages 549–603, Berlin.

Lim, A., Rodrigues, B., and Wang, Y. (2003). A multi-faced buildup algorithm for three-dimensional packing problems. *OMEGAInternat. J. of Management Sci.*, 31(6):471–481.

Lim, A. and Zhang, X. (2005). The container loading problem. In *Proc. of the 2005 ACM symposium on Applied computing (SAC'05)*, pages 913–917, New York, NY, USA. ACM Press.

Macedo, R., Alves, C., and Valerio de Carvalho, J. M. (2010). Arc-flow model for the two-dimensional guillotine cutting stock problem. *Computers and Operations Research*, 37(6):991–1001.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, Chichester.

Martello, S. and Vigo, D. (1998). Exact solution of the two-dimensional finite bin packing problem. *Management Sci.*, 44(3):388–399.

Ngoi, B. K. A., Tay, M. L., and Chua, E. S. (1994). Applying spatial representation techniques to the container packing problem. *Internat. J. of Production Res.*, 32(1):111–123.

O'Neill, M., Cleary, R., and Nikolov, N. (2004). Solving knapsack problems with attribute grammars. In *Proc. of the Third Grammatical Evolution Workshop (GEWS'04)*, Seattle, WA, USA.

Rhee, W. T. and Talagrand, M. (1993). On line bin packing with items of random size. *Math. Oper. Res.*, 18:438–445.

Richey, M. B. (1991). Improved bounds for harmonic-based bin packing algorithms. *Discrete Applied Mathematics*, 34:203–227.

Ross, P. (2005). Hyper-heuristics. In Burke, E. K. and Kendall, G., editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 529–556. Kluwer, Boston.

Ross, P., Marin-Blazquez, J. G., Schulenburg, S., and Hart, E. (2003). Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyperheurstics. In *Proc. of the Genetic and Evolutionary Computation Conf. 2003 (GECCO '03)*, pages 1295–1306, Chicago, Illinois.

Ross, P., Schulenburg, S., Marin-Blazquez, J. G., and Hart, E. (2002). Hyper heuristics: learning to combine simple heuristics in bin packing problems. In *Proc. of the Genetic and Evolutionary Computation Conf. 2002 (GECCO '02)*, New York, NY.

Schneider, W. (1988). Trim-loss minimization in a crepe-rubber mill; optimal solution versus heuristic in the 2 (3) - dimensional case. *Eur. J. of Oper. Res.*, 34(3):249–412.

Scholl, A., Klein, R., and Jurgens, C. (1997). Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Comput. and Oper. Res.*, 24(7):627–645.

Schwerin, P. and Wascher, G. (1997). The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *Internat. Trans. in Oper. Res.*, 4(5):377–389.

Seiden, S., Van Stee, R., and Epstein, L. (2003). New bounds for variable-sized online bin packing. *SIAM J. on Computing*, 32(2):455–469.

Terashima-Marin, H., Flores-Alvarez, E. J., and Ross, P. (2005). Hyper-heuristics and classifier systems for solving 2d-regular cutting stock problems. In *Proceeedings of the ACM Genetic and Evolutionary Computation Conf. (GECCO'05)*, pages 637–643, Washington, D.C. USA.

Terashima-Marin, H., Ross, P., Farias-Zarate, C. J., Lopez-Camacho, and Valenzuela-Rendon (2008). Generalized hyper-heuristics for solving 2d regular and irregular packing problems. *Annals of Operations Research*, Available online November 16.

Terashima-Marin, H., Zarate, C. J. F., Ross, P., and Valenzuela-Rendon, M. (2006). A ga-based method to produce generalized hyper-heuristics for the 2d-regular cutting stock problem. In *Proc. of the Genetic and Evolutionary Computation Conf. (GECCO'06)*, pages 591–598, Seattle, WA, USA.

Terashima-Marin, H., Zarate, C. J. F., Ross, P., and Valenzuela-Rendon, M. (2007). Comparing two models to generate hyper-heuristics for the 2d-regular bin-packing problem. In *Proc. of the Genetic and Evolutionary Computation Conf. (GECCO'07)*, pages 2182–2189, London, England.

Vasko, F. J., Wolf, F. E., and Stott, K. L. (1989). A practical solution to a fuzzy two-dimensional cutting stock problem. *Fuzzy Sets and Systems*, 29(3):259–275.

Yao, A. C.-C. (1980). New algorithms for bin packing. *J. of the ACM*, 27(2):207–227.

Yue, M. (1991). 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.