Evolving Reusable 3D Packing Heuristics with Genetic Programming

Sam Allen
sda@cs.nott.ac.uk

Edmund K. Burke
ekb@cs.nott.ac.uk

Matthew Hyde
mvh@cs.nott.ac.uk

Graham Kendall
gxk@cs.nott.ac.uk

The University of Nottingham
School of Computer Science
Nottingham, UK
NG8 1BB

ABSTRACT

This paper compares the quality of reusable heuristics designed by genetic programming (GP) to those designed by human programmers. The heuristics are designed for the three dimensional knapsack packing problem. Evolutionary computation has been employed many times to search for good quality solutions to such problems. However, actually *designing* heuristics with GP for this problem domain has never been investigated before. In contrast, the literature shows that it has taken years of experience by human analysts to design the very effective heuristic methods that currently exist.

Hyper-heuristics search a space of heuristics, rather than directly searching a solution space. GP operates as a hyper-heuristic in this paper, because it searches the space of heuristics that can be constructed from a given set of components. We show that GP can design simple, yet effective, stand-alone constructive heuristics. While these heuristics do not represent the best in the literature, the fact that they are designed by evolutionary computation, and are human competitive, provides evidence that further improvements in this GP methodology could yield heuristics superior to those designed by humans.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Design, Performance, Reliability

Keywords

Knapsack Packing, GP, Hyper-Heuristics, Heuristics

1. INTRODUCTION

The three dimensional knapsack packing problem is well known, and has many real world applications. There are many optimisation methodologies in the literature that report close to optimal results on benchmark instances. All of these methodologies operate by searching the space of solutions in an attempt to find the global optimum. In this paper we propose the first methodology for this problem that searches a space of constructive heuristics instead of searching the solution space directly. To achieve this, heuristics are evolved, utilising genetic programming (GP). The advantage of such a methodology is that the output is an automatically designed heuristic, that has been tailored to a given type of problem without human input. We show that the heuristic maintains human competitive performance on new unseen problem instances.

Burke et al. [5, 6, 7] have shown that it is possible to evolve stand-alone heuristics for one dimensional bin packing. The heuristics are shown to be superior to the human designed 'best-fit' heuristic on unseen problem instances from the class that they were evolved on. Pisinger and Egeblad proposed five classes of three dimensional problem instances [25], which appear ideal for use in a similar experiment for evolving three dimensional packing heuristics. This is the aim of this paper; to automatically design stand-alone heuristics through evolution, and then test their quality on new unseen problem instances with similar properties to those in their training set.

We show that GP can be used to evolve reusable packing heuristics which are competitive with both a state of the art deterministic constructive heuristic, and a simulated annealing metaheuristic methodology. This is a significant achievement, as it shows that an automatically designed heuristic can compete with two heuristics designed by humans. The results of this paper can be thought of as a comparison between a human programmer and an evolutionary computation methodology.

1.1 Hyper-Heuristics

A hyper-heuristic is defined as a heuristic which searches a space of heuristics, as opposed to a space of problem solutions [4, 27]. There are (at least) two classes of hyper-heuristic. One class aims to intelligently choose heuristics from a set of heuristics which have been provided. The other

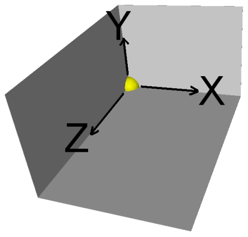

Figure 1: An initialised bin with one 'corner' in the back-left-bottom corner of the bin

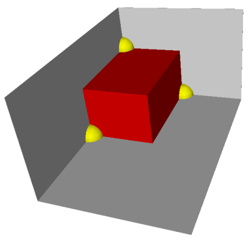

Figure 2: A bin with one piece placed in the corner from figure 1

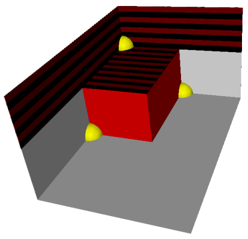

Figure 3: The three surfaces defined by the corner that is in the Y direction of the piece

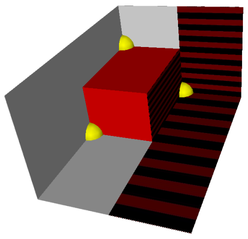

Figure 4: The three surfaces defined by the corner that is in the X direction of the piece

class aims to automatically generate heuristics from a set of components.

The first class has been implemented in the majority of previous hyper-heuristic work. Many metaheuristics and other machine learning techniques have been used as hyper-heuristics of this type. For example, simulated annealing [11], case based reasoning [9], and tabu search [8]. In contrast to the majority of previous hyper-heuristic work, this paper implements the second hyper-heuristic class. We use GP as a hyper-heuristic, to evolve new heuristics for a given set of problems. The GP is a hyper-heuristic because it searches a space of *all* the heuristics that can be created from the given components that we define, rather than searching a small static set of pre-defined heuristics. Previous work on this class of hyper-heuristic exists for the SAT problem [14, 15, 16], the travelling salesman problem [20], and evolving dispatching rules for the job shop problem [17]. Also, see [5, 6, 7] which report using GP as a hyper-heuristic to evolve reusable heuristics for the one dimensional bin packing problem.

1.2 Three Dimensional Knapsack Packing

The three dimensional knapsack problem consists of a set of pieces, a subset of which must be packed into a three dimensional container (the knapsack). Each piece has a size in three dimensions, and a separate 'value' (profit), which is independent of the piece's size. Often, in practice, the piece's value will be set equal to its volume. The objective is to maximise the value of the pieces chosen to be packed into the knapsack, without any overlap of other pieces or with the container edge. A full integer programming formulation of the problem is given in Egeblad and Pisinger [12]. In this paper, we allow 90° piece rotations in all cases, so each piece has six distinct orientations.

An intuitive heuristic procedure is described in [24], which constructs a solution by placing a piece in the position which results in the least wasted space around it. Chua et al. also use a similar spatial representation technique [10]. The genetic algorithm presented in [3] 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. present a 'multi-faced buildup' method [22]. Pisinger and Egeblad [25] employ a simulated annealing heuristic approach using 'sequence triples' as a representation. They also define the problem instance classes that are used in this paper. Huang and He [18] present a packing algorithm which uses a concept called 'caving degree', a measure of how close a box is to those already packed. In addition to these references, some heuristics and metaheuristics for the related three dimensional bin packing problem are given in [2, 13, 19].

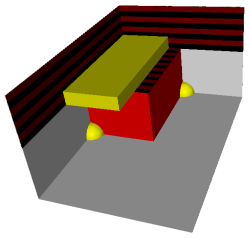

Figure 5: An invalid placement of a new piece, because it exceeds the limit of the corner's XZ surface in the Z direction

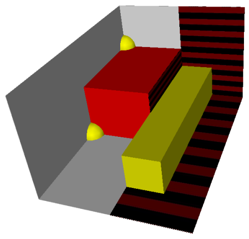

Figure 6: An invalid placement of a new piece, because it exceeds the limit of the corner's YZ surface in the Z direction

2. REPRESENTATION

This section describes the representation that the evolved heuristics employ to build the solutions. This choice of representation feeds directly into our choice of GP terminals.

2.1 Corner Objects

The knapsack is represented by its dimensions, and by a list of 'corner' objects, which represent the available spaces into which a piece can be placed. This is shown formally in equations 1 and 2.

$$B = \{w, h, d, L\} \tag{1}$$

$$L = \{C_1, \ldots, C_n\} \tag{2}$$

In equations 1 and 2, w, h, and d are the width, height, and depth of the knapsack, respectively. L represents a dynamically changing list of the corners in the knapsack, and n is equal to the number of corner objects currently available in knapsack B.

The knapsack is initialised by creating a corner which represents the lower-back-left corner of the knapsack, 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.

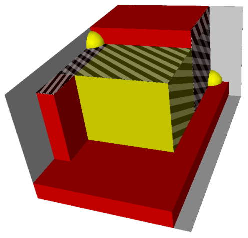

Figure 7: A piece which reaches the limit of two surfaces of the corner that it was put into

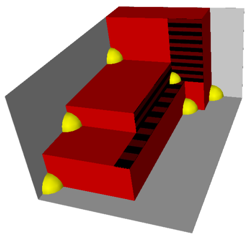

Figure 8: The three surfaces of the corner with the smallest avaliable area

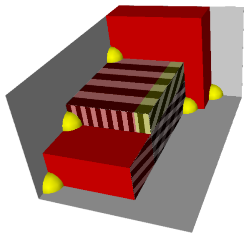

Figure 9: A filler piece is put into the corner from 8, the surfaces of two corners are updated

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 define it, in the XY plane, the XZ plane, and the YZ plane. The corner is at the intersection of these three orthogonal planes. Figure 3 shows the three 2D surfaces that the corner above the piece is defined by. Note that the XZ surface has its limits at the edges of the top of the piece. Similarly, figure 4 shows the three surfaces of the corner that is to the right of the piece in the figure.

2.2 Valid Placement of Pieces

We impose the following placement constraints on the packing process. 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.

2.3 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 is placed in the middle and the two corners shown must have their surfaces updated. In this situation, as is often the case, the piece does not extend an existing surface, but creates a new one in the same plane, that overlaps the existing surfaces. So the corner on the left of figure 7 now has two surfaces in its XZ plane, and the corner on the right has two surfaces in its YZ plane.

A formal representation of a corner is given in equations 3-7.

$$C_j = \{x_j, y_j, z_j, S_{xy}, S_{xz}, S_{yz}\}, \qquad j \in \{1, \dots, n\} \tag{3}$$

In equation 3, x_j, y_j, and z_j represent the coordinates of corner j in the bin. As stated in section 2.1, a corner contains information about the three 2D surfaces that define its position. The information about the three planes is represented by the three sets of surfaces, denoted by S.

$$S_{xy} = \{s_1, \dots, s_{m_{xy}}\} \tag{4}$$
$$S_{xz} = \{s_1, \dots, s_{m_{xz}}\} \tag{5}$$
$$S_{yz} = \{s_1, \dots, s_{m_{yz}}\} \tag{6}$$

In equations 4-6, m_{ab} is the number of distinct overlapping surfaces in the ab plane. For example, the corner on the right in figure 7 will have an m_{yz} value of 2, and the corner on the left hand side of figure 7 will have an m_{xz} value of 2. This is because there are now two distinct overlapping surfaces on each of these planes. Each surface has a length along each of the two axes of the plane to which it belongs. This is shown formally in equation 7, where l_a is the length of the surface s_k in the a dimension, and l_b is the length of the surface s_k in the b dimension.

$$s_k = \{l_a, l_b\}, \qquad k \in \{1, \dots, m_{ab}\} \tag{7}$$

2.4 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 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, so 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. 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. GP METHODOLOGY

In this paper, the heuristics we evolve are the individuals in the GP population, and are represented as tree structures. A heuristic decides both which piece to place next and where to put it. This section explains this decision making process. We will refer to a combination of piece, orientation and corner as an 'allocation'.

3.1 Evolving the Choice of Corner

The heuristic operates within a fixed framework, which results in the pieces being placed one at a time into one of the available corners. The framework evaluates the heuristic for each valid allocation that is available. The valid allocations are all of the combinations of each piece, corner, and orientation that do not result in the piece exceeding any surface of the corner. The heuristic is evaluated by setting the values of its terminals (explained in section 3.2) using the properties of the allocation's piece and corner. When the heuristic is evaluated, it returns one numerical value from its root node, which can be interpreted as the 'score' that the heuristic gives to the allocation.

After every valid allocation has been evaluated by the heuristic, the allocation which obtained the highest score is the one that is actually performed on the partial solution. The whole process then repeats in order to pack another piece. In this way, the heuristic does not rely on a good piece ordering, because any piece can be chosen from all of those yet to be packed.

Before the heuristic starts evaluating all of the available allocations, all of the corners are checked to see if they are too small to accommodate any of the remaining pieces. If they are, then a filler piece is placed into the corner as explained in section 2.4.

3.2 GP Terminals

The functions and terminals are shown in table 1. We use four functions, add, subtract, multiply and protected divide.

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

Table 2: Parameters of each GP 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

Our protected divide function replaces the denominator by 0.001 if it is zero. There are also nine terminals. The volume of the piece, and the value of the piece are both represented by terminals. There are three 'waste' terminals which represent how well the piece fits into the corner in its current orientation, and three terminals which can give the heuristic the position of the corner in the knapsack.

3.3 GP Parameters

Table 2 shows the parameters used in the GP runs. The mutation operator is point mutation, using the 'grow' method explained in [21], 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 ECJ (Evolutionary Computation in Java, http://cs.gmu.edu/∼eclab/projects/ecj/) package we utilised for our implementation. We also apply a parsimony pressure to the population, in the form of a Tarpeian Wrapper [26], which ignores 20% of the individuals which have above average size.

3.4 Fitness Measure

During the GP run, a heuristic is evaluated by using it to pack the ten training instances for one particular class. The fitness for each instance is the total value of the pieces in

Table 3: The classes of the EP3D instances, and the dimensions of the pieces

Class	Description	Width	Height	Depth
F	Flat	[50,100]	[50,100]	[25,60]
L	Long	[1,$\frac{2}{3}$.100]	[1,$\frac{2}{3}$.100]	[50,100]
C	Cubes	[1,100]	[=Width]	[=Width]
U	Uniform	[50,100]	[50,100]	[50,100]
D	Diverse	[1,50]	[1,50]	[1,50]

the knapsack. This is shown mathematically in equation 8, where n is the number of pieces, v_p is the value of piece p, and x_p is a binary variable indicating if piece p is packed in the knapsack. The heuristic receives a fitness equal to the sum of the fitnesses it obtains on the ten instances.

$$instancefitness = \sum_{p=1}^{n} v_p x_p \qquad (8)$$

4. PROBLEM INSTANCES

The problem instances used to evolve and test the heuristics all contain 40 pieces. They can be separated into five main classes, defined in [12, 25]. Each of the five main classes consists of either flat, long, cube, uniform, or diverse pieces, these classes are shown in table 3. In each of these main classes there are four subclasses, two contain 'clustered' pieces (five piece sizes are generated, and eight pieces are included from each), and two contain 40 pieces generated independently. One of each of these two has its knapsack volume at 50% of the total volume of pieces, and the other one has its knapsack volume at 90%. Therefore, there are 20 distinct classes in total, four from each of the five main classes. The dimensions of the knapsack are always set so that the height and width are equal, and the depth is twice the size of the width. The exact dimensions are determined by whether the knasack volume must be 50% or 90% of the total volume of the pieces.

We will use the notation [X,Y,Z] to denote a class, where X, Y and Z represent features of the class. X is the general shape of the pieces, and can be either C, D, F, L, or U, representing flat, long, cube, uniform, or diverse pieces. Y is either C or R, for clustered or random. Z can be either 50 or 90, representing the volume of the knapsack relative to the total volume of the pieces. For example, [F,R,50] is the class with flat pieces, which are not clustered, and the knapsack always has a volume of 50% of the total volume of pieces.

We generate two instance sets, the first is used for experient one, and the second is used for experiment two. Each instance set contains 10 training instances per class, on which to evolve the heuristics. In addition to this, the first set contains 10 further instances per class, on which to test the heuristics once they are evolved. So, in the first set, there are 20 instances in total per class, 10 for evolving the heuristics, and 10 for testing the quality of the evolved heuristic. In addition to its 10 training instances per class, the second set contains the exact 20 instances (one from each class) that are used in [25], and these are used to test the evolved heuristics. So, in the second set, there are 11 instances in total per class, 10 for evolving the heuristics, and one for testing the quality of the evolved heuristic.

For the first instance set, the value of a piece is set to be equal to its volume. This is in contrast to the instances of the second set, and to those generated in [12, 25]. These instances are used in this paper to allow a comparison between the evolved heuristics and the human created 3BF heuristic, which does not cater for instances where a piece's value is different to its volume. In the second instance set, the pieces have a value 200 units greater than their volume. This is identical to the method of generating instances in [25]. These instances are used so that a comparison can be made with the results of the STSA algorithm from [25]. The problem instance generator used in [25] is available at: http://www.diku.dk/~pisinger/codes.html

As previously stated, for both instance sets, we generate 10 training instances from each of the 20 classes (generating 200 instances in total), in order to evolve the heuristics. Each heuristic is evolved using the fitness measure (see section 3.4) derived from its performance over the 10 training instances of one of the 20 classes. We perform ten GP runs on each class, each run produces a heuristic, making 200 heuristics in total. For all instances, we allow 90° piece rotations, so each piece has six possible orientations.

5. RESULTS

5.1 The 3BF Heuristic

The 3BF heuristic is an extremely effective heuristic packing algorithm, which achieves some of the best results in the literature [1]. It is constructive and deterministic, which makes it ideal to compare against, as our GP methodology also designs constructive deterministic heuristics. For a full explanation of 3BF, please refer to [1], but a brief description is given here for ease of reference.

3BF packs pieces one at a time, each into the lowest available continuous surface in the current partial solution. Before any pieces have been packed, the lowest available surface has a corner at [0,0,0], and stretches to the edges of the knapsack. The location and dimensions of the lowest surface will change dynamically during the packing, as more pieces are added. Any piece can be chosen to be placed onto the surface from those still waiting to be packed, so the method does not rely on the pieces initially being in a 'good order'.

Once the lowest surface has been identified, all of the remaining pieces are assessed as to how well they fit onto it in all orientations. The heuristic chooses the piece that covers the greatest area of the surface. If two or more pieces tie, then the heuristic chooses between them by applying one of four placement strategies, one of which is chosen before the packing begins. These consist of; putting the largest piece into the bottom-leftmost corner, putting the smallest piece into the bottom-leftmost corner, maximising the total contact with surrounding pieces, and lining up a face of the box with faces of surrounding boxes (to create continuous 'walls' in front of which other pieces can be placed).

5.2 Results on Problem Set One

In this section, we compare the results of our evolved heuristics against the results of the 3BF constructive algorithm of Allen et al. [1]. See section 5.1 for an explanation of this heuristic. The comparison of results is shown in table 4. The evolved heuristics are compared with 3BF on the ten instances from each class which are unseen during the evolution phase of the heuristics. The '% Diff' column of table 4 shows the percentage difference between the results of our evolved heuristics and the result of 3BF. Therefore, a value of less than zero means the evolved heuristics performed better.

Ten heuristics were evolved for each class, in the ten runs of the GP. Each row of the table summarises the results of the *one* evolved heuristic which performed best on the ten training instances of that class. The figure reported in each row is the average of the results of that heuristic over the ten test instances of the class. The figure reported for 3BF is also the average of its performance over the ten test instances of the class.

For each class, we obtain 100 results, ten results for each of ten heuristics. There are therefore ten results for each instance, one from each heuristic. The performance of any given heuristic varies over the ten instances of the class, and for each of the ten instances some heuristics are better than others. For example, heuristic one may be better than heuristic seven on instances 1-3, and heuristic seven may be better than heuristic one on instances 6-9. To obtain the average reported in table 4 for this class, we do not use the best result for each individual instance regardless of which heuristic achieved it. Rather, we use the full ten results from only *one* heuristic. This is because we want each row of the table to reflect a comparison between 3BF and just *one* automatically generated heuristic, not between 3BF and a combination of the results of ten evolved heuristics.

The results of the evolved heuristics are all within 19% of the results of 3BF, with the majority being within 10%. The mean average of the differences is 9%. Importantly, these results show that a heuristic that is automatically designed, with no human input, can perform within a small percentage of the performance of the best human designed constructive algorithm for certain classes of this problem.

The heuristic evolved on the [C-R-50] class actually performs better than 3BF on new instances, even though it is designed solely by GP. This is achieved on new problem instances, that were not seen by the GP during its design. It is important to also note the limited set of ten problem instances, upon which the GP had to rely to produce a general heuristic for similar problems. Also note that 3BF makes use of four human designed placement strategies, and returns the best one (see section 5.1). This represents four separate attempts at the packing, using a different strategy each time. An evolved heuristic essentially represents just one combined strategy for selection and placement, and only performs one packing.

While not shown in the table due to space constraints, the evolved heuristics perform much better on their training instances. The heuristics all perform within 8% of 3BF, with a mean average of 3.1%. Three heuristics are evolved to be better than 3BF on their training sets. This shows that heuristics which pack to a high standard can be evolved, but future work must include improving the generality of the heuristics.

Human designers can rely on preconceptions, experience, and analysis of the domain in order to produce a good heuristic for a set of problems. GP does not have this luxury, and automatically designing simple heuristics only a few percent worse than 3BF, from a set of only ten training instances, is a significant result. This is especially true of the 'Diverse' class of instances, where the piece dimensions vary greatly between instances, and it would intuitively seem quite difficult to generalise from only ten of these training instances.

Table 4: The results of experiment one. The table compares the best of the ten heuristics evolved for each class, to the human created 3BF heurisitic. For the evolved heuristics and the 3BF heuristic, the table reports the average result over the ten test instances for each class.

Class	3BF	Evolved	% Diff
[C-C-50]	3733532.8	3443139.1	7.8
[C-C-90]	7455604.1	7292573.1	2.2
[C-R-50]	3430999.1	3455112.4	-0.7
[C-R-90]	7060685.3	6296610.9	10.8
[D-C-50]	344402.9	308466.7	10.4
[D-C-90]	592606	518633	12.5
[D-R-50]	313272.2	301923.2	3.6
[D-R-90]	551588.4	537832.2	2.5
[F-C-50]	4005811.3	3283458.9	18.0
[F-C-90]	7233309.1	6215685.8	14.1
[F-R-50]	4250967	3753119	11.7
[F-R-90]	7637004.3	6872005.3	10.0
[L-C-50]	1273426	1040132.7	18.3
[L-C-90]	2316755.9	1877989.4	18.9
[L-R-50]	1439979.5	1361067.2	5.5
[L-R-90]	2541315.7	2301376.2	9.4
[U-C-50]	6946577.1	6425425.6	7.5
[U-C-90]	11858492.6	11180196.4	5.7
[U-R-50]	7899483.7	7303117.2	7.5
[U-R-90]	12995621.6	12438440.5	4.3

5.3 The STSA Algorithm

The sequence triple simulated annealing algorithm (STSA) is presented by Egeblad and Pisinger. We compare the results of the evolved heuristics against STSA in section 5.4. For a full explanation of STSA, please refer to [12, 25], but a brief explanation is given here for ease of reference. The algorithm representation is based on the sequence pair representation that Murata et al. used for two dimensional packing [23]. The sequence triple representation consists of three related lists, each containing all of the items. The relative positions of two items in the lists determine their relative positions in the packing. For example, if piece $p1$ is before $p2$ in lists one and two, but not in list three, then $p1$ is positioned over $p2$. If piece $p1$ is before $p2$ in lists two and three, but not in list one, then $p1$ is positioned to the right of $p2$. See [25] for the full list of these rules. A neighbourhood move in the simulated annealing algorithm is defined as an exchange of the positions of two items in one or more lists.

5.4 Results on Problem Set Two

In this section, we compare the results of evolved heuristics against the results of the sequence triple simulated annealing algorithm (STSA) by Egeblad and Pisinger [12, 25]. The comparison of results is shown in table 5. They are compared by packing the instances taken from [25], which are unseen during the evolution phase of the heuristics. The STSA results in the table are taken directly from [25].

The '% Diff' column of table 5 shows the percentage difference between the results of our evolved heuristics and result of STSA. A value greater than zero means the evolved heuristics performed better. Each row of the table summarises the results of the ten heuristics that were evolved

Table 5: The results of experiment two, the evolved heuristics compared to the performance of the sequence triple simulated annealing algorithm (STSA), on the 20 test instances of instance set two

	STSA	Evolved	% Diff
ep3d-40-C-C-50	1265664	1265664	-0.0
ep3d-40-C-C-90	2828160	2828160	-0.0
ep3d-40-C-R-50	2760843.9	2699658.7	2.2
ep3d-40-C-R-90	5937447.5	5253038.1	11.5
ep3d-40-D-C-50	525116.4	632605.6	-20.5
ep3d-40-D-C-90	1124263.6	1152560.8	-2.5
ep3d-40-D-R-50	338144.5	328910.8	2.7
ep3d-40-D-R-90	612172.9	557471.1	8.9
ep3d-40-F-C-50	3538845.4	3367094.1	4.9
ep3d-40-F-C-90	6158772.9	7037600.4	-14.3
ep3d-40-F-R-50	3407281.7	3683234.9	-8.1
ep3d-40-F-R-90	7233223.4	6921811.6	4.3
ep3d-40-L-C-50	1659816.2	1809067.8	-9.0
ep3d-40-L-C-90	2815563.7	2978586.2	-5.8
ep3d-40-L-R-50	1579902.6	1577057	0.2
ep3d-40-L-R-90	2618748	2643232.2	-0.9
ep3d-40-U-C-50	7008136	7114938.8	-1.5
ep3d-40-U-C-90	13761564.4	12269377.2	10.8
ep3d-40-U-R-50	7653893.4	7203887.2	5.9
ep3d-40-U-R-90	12759327	11480013	10.0

for that class, on the one test instance from that class (the one used in [25]). The row shows the average results of the ten heuristics evolved for that class, and compares this to the average results from Pisinger and Egeblad's STSA algorithm [25] over ten runs.

The results of the evolved heuristics are highly competitive with the results of STSA, and have a better average in some cases. Ten of the average results are better than, or equal to, STSA. Also, while not shown in the table, in twelve classes the best of the ten heuristics is better than or equal to the best result obtained by STSA. This shows that the quality of the evolved heuristics is highly competitive with this human created metaheuristic.

Subsequent work by Egeblad and Pisinger in [12] uses a slightly different problem instance generator, which sets the value of a piece to be equal to the width multiplied by the height, multiplied by a random integer between one and three inclusive. Because the pieces' depth is ignored, we have had less success in evolving generalisable heuristics on these instances. This is the reason for using the problem instance generator used in [25] instead, as training on such instances can produce generalisable heuristics.

6. CONCLUSIONS

It is not a goal of this paper to present the results of this GP methodology as the best in the literature for three dimensional knapsack packing. Clearly they are not the best results for these problem instances, and as we have shown, the 3BF heuristic nearly always produces better results. However, this heuristic was created by a human programmer, and the evolved heuristics are created by GP. Currently, we cannot reasonably expect that GP will exceed human expertise in the fields of problem analysis and heuristic design for three dimensional knapsack packing. The contribution of this paper is to show that GP can design stand-alone heuristics which are at least competitive with human created heuristics, and, of course, we have also shown that some evolved heuristics *are* in fact better on some classes of problem instances. In particular, the evolved heuristics are highly competitive with the metaheuristic STSA algorithm, even though that is a sophisticated algorithm, and the evolved heuristics are simple, constructive, and deterministic.

Related work in [6] showed that, in one dimensional bin packing, heuristics can be evolved in much the same way, for a class of problem instances, and then reused on problems only of the same class. When tested on problems of a different class, the evolved heuristics often perform very poorly. This is because they must pack pieces with sizes that they have not seen before during their evolution phase. The work presented in this paper has only tested the evolved heuristics on new instances of the same class, but we hypothesise that the heuristics would perform less well if they were tested on problem instances of a different class to that which they were evolved for. This is a hypothesis we intend to test in further work, to investigate if the conclusions of [6] also hold for three dimensional packing.

Even with this limitation, automatic heuristic generation is still valuable, in situations where human created heuristics are expensive, and where the heuristic is only likely to be asked to solve problems similar to those it was trained on. Though it does mean that the user must be highly selective which problem instances are included in the training set of the heuristic. If the training set is representative of all the instances that the heuristic will solve in the future, then a reusable heuristic can be generated and tailored automatically. This automation potentially means it would be cheaper to create than it would be to employ a human analyst to create a packing heuristic.

We intend to investigate three further aspects of this research. Firstly, we will investigate the effect of increasing and decreasing the number of instances in the training set. Secondly, we will investigate if good quality heuristic generation is easier, or more difficult, for instances smaller and larger than the 40 piece instances used here. Thirdly, we will investigate if the evolved heuristics can maintain their quality when applied to instances with many more pieces than than the instances in its training set. If this is the case, then the computationally intensive evolution phase could be executed with scaled down instances to save time. This would be similar to research into the scalability of evolved one dimensional bin packing heuristics by Burke et al. [7].

This is the first paper to present automatically generated heuristics for this problem, and we aim to contribute to the growing literature on automatic heuristic generation [6, 7, 15, 16, 17, 20]. In this literature, the results of generated heuristics have often been shown to be better than human created heuristics. But, in this early stage of research in this field, the aim is to research methodologies to automate the design process itself, not necessarily to obtain the best results. The evidence here suggests that if the GP methodology is improved upon, or other techniques are introduced, then the field of automatic heuristic generation has the potential to *routinely* produce heuristics which are superior to those created by humans for this difficult problem.

7. REFERENCES

[1] S. Allen, E. K. Burke, and G. Kendall. A new hybrid placement strategy for the three-dimensional strip packing problem. Technical report, University of Nottingham, Dept of Computer Science, 2009.

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

[3] A. Bortfeldt and H. Gehring. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131(1):143–161, 2001.

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

[5] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *LNCS 4193, Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006)*, pages 860–869, 2006.

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

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

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

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

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

[11] K. Dowsland, E. Soubeiga, and E. K. Burke. A simulated annealing hyper-heuristic for determining shipper sizes. *European Journal of Operational Research*, 179(3):759–774, 2007.

[12] J. Egeblad and D. Pisinger. Heuristic approaches for the two- and three-dimensional knapsack packing problem. *Computers and Operations Research*, 36(4):1026–1049, 2009.

[13] M. Eley. Solving container loading problems by block arrangement. *European Journal of Operational Research*, 141(2):393–409, 2002.

[14] A. S. Fukunaga. Automated discovery of composite sat variable-selection heuristics. In *Eighteenth national conference on Artificial intelligence*, pages 641–648, Menlo Park, CA, USA, 2002.

[15] A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *LNCS 3103. Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO '04)*, pages 483–494, Seattle, WA, USA, 2004. Springer-Verlag.

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

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

[18] W. Huang and K. He. A new heuristic algorithm for cuboids packing with no orientation constraints. *Computers and Operations Research, In Press, Corrected Proof, Available online 21 September 2007.*

[19] N. Ivancic, K. Mathur, and B. Mohanty. An integer-programming based heuristic approach to the three-dimensional packing problem. *Journal of Manufacturing and Operations Management*, 2:268–298, 1989.

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

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

[22] A. Lim, B. Rodrigues, and Y. Wang. A multi-faced buildup algorithm for three-dimensional packing problems. *OMEGA International Journal of Management Science*, 31(6):471–481, 2003.

[23] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Vlsi module placement based on rectangle-packing by the sequence-pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1518–1524, 1996.

[24] B. K. A. Ngoi, M. L. Tay, and E. S. Chua. Applying spatial representation techniques to the container packing problem. *International Journal of Production Research*, 32(1):111–123, 1994.

[25] D. Pisinger and J. Egeblad. Heuristic approaches for the two and three dimensional knapsack packing problems. Technical report, No.06-13. University of Copenhagen, Dept of Computer Science, 2006.

[26] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, pages 211–223, Essex, April 2003. Springer-Verlag.

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