A Genetic Programming Hyper-Heuristic Approach for Evolving

Edmund Burke, Matthew Hyde
Graham Kendall and John Woodward

First released: 2008
© Copyright 2008 Edmund Burke, Matthew Hyde, Graham Kendall and John Woodward
A Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics

Edmund K Burke, Member, IEEE, Matthew Hyde, Graham Kendall, Member, IEEE, and John Woodward

Abstract—We present a genetic programming hyper-heuristic system to evolve a 'disposable' heuristic for each of a wide range of benchmark instances of the two-dimensional strip packing problem. The evolved heuristics are constructive, and decide both which piece to pack next and where to place that piece, given the current partial solution. Usually, there is a trade-off between the generality of a packing heuristic and its performance on a specific instance. This hyper-heuristic approach is particularly beneficial because it produces a heuristic tuned to the instance, and it is general enough that it can do so for any strip packing instance, with no change of parameters. The contribution of this paper is to show that the genetic programming hyper-heuristic can be employed to automatically generate heuristics which are better than the human-designed state of the art constructive heuristics, in a very well studied area.

Index Terms—Genetic programming, hyper-heuristics, two-dimensional stock cutting

I. INTRODUCTION

This paper presents a hyper-heuristic genetic programming system for the two-dimensional strip packing problem, where a number of rectangles must be placed onto a sheet with the objective of minimising the length of sheet that is required to accommodate the items. The sheet has a fixed width, and the required length of the sheet is measured as the distance from the base of the sheet to the piece edge furthest from the base. This problem is known to be NP hard [1], and has many industrial applications as there are many situations where a series of rectangles of different sizes must be cut from a sheet of material (for example, glass or metal) while minimising waste.

Indeed, many industrial problems are not limited to just rectangles (for example textiles, leather, etc) and this presents another challenging problem [2]. There are many other types of cutting and packing problems in one, two and three dimensions. A typology of these problems is presented by Wascher et al. in [3]. As well as their dimensionality, the problems are further classified into different types of knapsack and bin packing problems, and by how similar the pieces are to each other.

In this work, the rectangular pieces are free to rotate by 90 degrees, and there can be no overlap of pieces. The guillotine version of this problem occurs where the cuts to the material can only be made perpendicular to an edge, and must split the sheet into two pieces, then those same cutting rules apply recursively to each piece. However, we are interested here in the non-guillotine version of the problem, which has no such constraint on how the pieces are cut.

The motivation behind this work is to develop a hyper-heuristic genetic programming methodology which can automatically generate a novel heuristic for any new problem instance. This has the potential to eliminate the time consuming process of manual problem analysis and heuristic building that a human programmer would carry out when faced with a new problem instance or set of instances. Work on automatic heuristic generation has not been presented before for this problem. However, similar work on other problem domains has been published (see section II-B).

Hyper-heuristics are defined as heuristics which search a space of heuristics, as opposed to searching a space of solutions directly [4] (which, of course, is the conventional approach to employing evolutionary algorithms). We employ genetic programming as a hyper-heuristic to search the space of heuristics that it is possible to construct from a set of building blocks. The output is a heuristic which is applied to the solution space to produce a solution.

We show that our hyper-heuristic designs and builds heuristics which are competitive with human designed heuristic and metaheuristic approaches. Also, the system is general enough to do so without any change in parameters. Any problem instance can be given to the hyper-heuristic and a heuristic will be produced that obtains a solution.

There are two components to any constructive heuristic used for the two dimensional strip packing problem. Many of the heuristics created by humans are reliant on the presented order of the pieces before the packing begins. Often, the pieces are pre-ordered by size, which can achieve better results [5]. However, it is not currently possible to say that this will result in a better packing, in the general case, than a random ordering.

Metaheuristics have been successfully employed to generate a good ordering of the pieces before using a simple placement policy to pack them [6], [5]. These hybrid metaheuristic approaches have shown that it is possible for one heuristic to gain good results on a wider range of instances because of the ability to evolve a specific ordering of the pieces for a given instance. However, they are still limited by the fact that their packing heuristic may not perform well on the instance regardless of the piece order, which would make it difficult for the hybrid approach to find a good solution.

The heuristics we evolve do not suffer from the same limitations. As we will show, they decide which piece to place next in the partial solution and where to place it. So
the evolved heuristics’ performance is independent of any piece order.

In section II, we introduce the background literature on genetic programming, hyper-heuristics, and 2D strip packing approaches. Section III presents our algorithm. Section IV describes the benchmark problem instances used in this paper, and section V presents the results we have obtained for those benchmark instances and compares them to recent results in the literature. Finally, conclusions and ideas for future work are given.

II. BACKGROUND

A. Genetic Programming

Genetic programming (GP) (see [7], [8], [9]) is a technique used to evolve populations of computer programs represented as tree structures. An individual’s performance is assessed by evaluating its performance at a specific task, and genetic operators such as crossover and mutation are performed on the individuals between generations. A list of GP parameters used in this paper is given in section III-D.

B. Hyper-Heuristics

Hyper-heuristics are defined as heuristics that search a space of heuristics, as opposed to searching a space of solutions directly [4]. Research in this area is motivated by the goal of raising the level of generality at which optimisation systems can operate [10], and by the assertion that in many real-world problem domains, there are users who are interested in “good-enough, soon-enough, cheap-enough” solutions to their search problems, rather than optimal solutions [10]. In practice, this means researching systems that are capable of operating over a range of different problem instances and sometimes even across problem domains, without expensive manual parameter tuning, and while still maintaining a certain level of solution quality.

Many existing metaheuristics have been used successfully as hyper-heuristics. Both a genetic algorithm [11] and a learning classifier system [12] have been used as hyper-heuristics for the one-dimensional bin packing problem. A genetic algorithm with an adaptive length chromosome and a tabu list was used in [13] as a hyper-heuristic. A case based reasoning hyper-heuristic is used in [14] for both exam timetabling and course timetabling. Simulated annealing is employed as a hyper-heuristic in [15] for the shipper rationalisation problem. A tabu search hyper-heuristic [16] is shown to be general enough to be applied to two very different domains: nurse scheduling and university course timetabling. A graph based hyper-heuristic for timetabling problems is presented in [17]. Three new hyper-heuristic architectures are presented in [18], treating mutational and hill climbing low-level heuristics separately. A choice function has also been employed as a hyper-heuristic, to rank the low-level heuristics and choose the best [19]. A distributed choice function hyper-heuristic is presented in [20].

The common theme to the hyper-heuristic research mentioned above is that all of the approaches are given a set of low-level heuristics, and the hyper-heuristic chooses the best one or the best sequence from those. Another class of hyper-heuristic, which has received less attention in the literature, generates low-level heuristics from a set of building blocks given to it by the user. The aim of this class of hyper-heuristic is to create a low-level heuristic from these building blocks, which will be either re-usable on new problems, or ‘disposable’, in the sense that they will only be evolved for the purpose of finding a solution to a particular instance. There are two main examples of this class of hyper-heuristic. The ‘CLASS’ system presented in [21],[22] and [23] is an automatic generator of local search heuristics for the SAT problem, and is competitive with human-designed heuristics. Simple one-dimensional bin packing heuristics are evolved in [24], which match the performance of the human-designed first-fit heuristic.

C. 2D Stock Cutting Approaches

1) Exact Methods: Gilmore and Gomory [25] first used a linear programming approach in 1961 to solve small size problem instances. Tree search procedures have been employed more recently to produce optimal solutions for small instances of the 2D guillotine stock cutting problem [26] and 2D non-guillotine stock cutting problem [27]. The method used in [26] has since been improved in [28] and [29]. It is recognised that these methods cannot provide good results for large instances. For example, the largest instance used in both [28] and [29] is 60 pieces. For larger instances, therefore, a heuristic approach must be introduced, which cannot guarantee the optimal solution, but can produce high quality results for much larger problems.

2) Heuristic Methods: Baker et al. define a class of packing algorithms named ‘bottom up, left justified’ (BL) [30]. These algorithms maintain bottom left stability during the construction of the solution, meaning that all the pieces cannot be moved any further down or left from where they are positioned. The heuristic presented in [30] has come to be named ‘bottom-left-fill’ (BLF) because it places each piece in turn into the lowest available position, including any ‘holes’ in the solution, and then left justifying it. While this heuristic is intuitively simple, implementations are often not efficient because of the difficulty in analysing the holes in the solution for the points that a piece can be placed [31]. Chazelle presents an optimal method for determining the ordered list of points that a piece can be put into, using a ‘spring’ representation to analyse the structure of the holes [31].

These heuristics take, as input, a list of pieces, and the results rely heavily on the pieces being in a ‘good’ order [30]. Theoretical work presented by Brown et al. [32] shows the lower bounds for online algorithms both for pre-ordered piece lists by decreasing height and width, and non pre-ordered lists. Results in [5] have shown that pre-ordering the pieces by decreasing width or decreasing height before applying BL or BLF results in performance increases of between 5% and 10%.
Recently, a ‘best-fit’ style heuristic was presented in [33]. 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 [33]. The best-fit heuristic packs the pieces in a similar way to the heuristics we evolve in this paper, because all the pieces are considered for packing, not just the first in the sequence given to it. However, in contrast, best-fit only considers one space for the pieces to go, and always packs the selected piece on a pre-determined side of the space. The heuristics evolved in this paper consider all spaces and both sides of each space.

Zhang et al [34] use a recursive algorithm, running in \(O(n^3)\) time, to create good strip packing solutions, based on the ‘divide and conquer’ principle. Finally, two heuristics for the strip cutting problem with sequencing constraint are presented by Rinaldi and Franz [35], based on a mixed integer linear programming formulation of the problem.

3) Metaheuristic Methods: Metaheuristics have been successfully employed to evolve a good ordering of pieces for a simple heuristic to pack. For example, Jakobs [6] uses a genetic algorithm to evolve a sequence of pieces for a simpler variant of the BL heuristic. This variant packs each piece by initially placing it in the top right of the sheet and repeating the cycle of moving it down as far as it will go, and then left as far as it will go. Liu and Teng [36] proposed a simple BL heuristic to use with a genetic algorithm that evolves the order of pieces. Their heuristic moves the piece down and to the left, but as soon as the piece can move down it is allowed to do so. However, using a BL approach with a metaheuristic to evolve the piece order is somewhat limited, for example it is shown in [30], [37] that, for certain instances, there is no sequence that can be given to the BLF heuristic that results in the optimal solution.

Ramesh Babu and Ramesh Babu [38] use a genetic algorithm in the same way as Jakobs, to evolve an order of pieces, but use a slightly different heuristic to pack the pieces, and different genetic algorithm parameters, improving on Jakobs’ results. Simulated annealing is used in [39], to evolve a piece order for the best-fit algorithm of [33].

Valenzuela and Wang [40] employ a genetic algorithm for the guillotine variant of the problem. They use a linear representation of a slicing tree as the chromosome. The slicing tree determines the order that the guillotine cuts are made and between which pieces. The slicing trees bear a similarity with the genetic programming trees in this paper, which represent heuristics. The slicing trees are not heuristics however, they only have relevance to the instance they are applied to, while a heuristic dynamically takes into account the piece sizes of an instance before making a judgement on where to place a piece. If the pattern of cuts dictated by the slicing tree were to be applied to a new instance, the pattern does not consider any properties of the new pieces. For example, if the slicing tree defines a cut between piece one and piece nine, then this cut will blindly be made in the new instance even if these pieces now have wildly different sizes. A heuristic would consider the piece sizes and the spaces available before making a decision.

Hopper and Turton [5] compare the performance of several metaheuristic approaches for evolving the piece order, each with both the BL constructive algorithm of Jakobs [6], and the BLF algorithm of [30]. Simulated annealing, a genetic algorithm, naive evolution, hill climbing, and random search are all evaluated on benchmark instances, and the results show that better results are obtained when the algorithms are combined with the BLF decoder. The genetic algorithm and BLF decoder (GA+BLF) and the simulated annealing approach with BLF decoder (SA+BLF) are used as benchmarks in this paper.

Other approaches start with a solution and iteratively improve it, rather than heuristically constructing a solution. Lai and Chan [41] and Faina [42] both use a simulated annealing approach in this way, and achieve good results on problems of small size. Also, Bortfeldt [43] uses a GA which operates directly on the representations of strip packing solutions.

A reactive greedy randomised adaptive search procedure (reactive GRASP) is presented in [44] for the two-dimensional strip packing problem. The method involves a constructive phase and a subsequent iterative improvement phase. To obtain the final overall algorithm, four parameters were chosen with the results from a computational study, using some of the problem sets used in the paper. First, one of four methods of selecting the next piece to pack is chosen. Second, a method of randomising the piece selection is chosen from a choice of four. Third, there are five options for choosing a parameter \(\delta\), which is used in the randomisation method, and finally there are four choices for the iterative improvement algorithm after the construction phase is complete. The method is a complex algorithm with many parameters, which are chosen by hand.

Belov et al. have obtained arguably the best results in the literature for this problem [37]. Their ‘SVC’ algorithm is based on an iterative process, repeatedly applying one constructive heuristic, ‘SubKP’, to the problem, each time updating certain parameters that guide its packing. The results obtained are very similar to those obtained by the GRASP method. They obtain the same overall result on the ‘C’, ‘N’ and ‘T’ instances of Hopper and Turton, but SVC obtains a slightly better result on ten instance sets from Berkey and Wang, and Martello and Vigo. Together, SVC(SubKP) and the reactive GRASP method represent the state of the art in the literature, and SVC(SubKP) seems to work better for larger instances [37].

We compare with the results of the reactive GRASP in section V, because they represent some of the best in the literature, and their reported results cover all of the data sets that we have used here. It must be noted however that the aims of the hyper-heuristic methodology presented in this paper differ slightly from the aims of other work in the literature. The aim here is to generate constructive heuristics
The quality of the results that the heuristics produce is of high importance, but we do not aim only for better results than the state of the art handcrafted heuristics. Therefore, the contribution of this paper is to show that automatically generated constructive heuristics can obtain results in the same region as the current state of the art human developed heuristics in two-dimensional strip packing [44], [37], which use a constructive phase and an iterative phase. We also show that the automatically generated constructive heuristics obtain better results than the human designed state of the art constructive heuristic, presented in [33].

III. METHODOLOGY

In section III-A, we explain the representation of the problem that we use, and how it is updated each time a piece is placed in the solution. Section III-B explains how the heuristic decides which piece to pack next and where to place it. A step by step packing example is given in section III-C to further clarify this process. Section III-D explains how the heuristics themselves are evolved, detailing the GP parameters.

A. Representation of the Problem

A constructive heuristic considers the strip packing problem to be a sequence of steps, where a piece must be placed at each step. Our hyper-heuristic system evolves a constructive heuristic for a given problem instance. At every step, the heuristic chooses a piece, and the position to place it, according to the state of the sheet and the pieces already placed on it. To this end, the sheet is represented as a set of dynamic ‘bins’, the number and configuration of which will change at every step. Each bin has a height (the distance from the base of the sheet to the base of the bin), a lateral position, and a width.

This representation takes its inspiration from the paper by Burke, Kendall and Whitwell [33], which introduces the best-fit heuristic for two dimensional packing. This heuristic selects the lowest available space, and then finds the piece which best fits it. This means the solution is essentially represented as a number of dynamic bins, in order to efficiently find the lowest one.

At the beginning of the packing process the sheet will be represented as just one bin, with height zero and width equal to the width of the sheet. As an example, figure 1 shows the bin configuration (three bins, with different heights and widths) when two pieces have been placed onto the sheet in the lower-right and lower-left corners. Figures 1-4 show a step by step example of two more pieces being packed and how the bin structure changes as each piece is packed. There follows a description of the four situations where the bin structure will change.

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

\[
\text{Height}(B_y) = \text{Height}(B_x) + \text{Height}(P)\\
\text{Height}(B_z) = \text{Height}(B_x)\\
\text{Width}(B_y) = \text{Width}(P)\\
\text{Width}(B_z) = \text{Width}(B_x) - \text{Width}(P)
\]

This process is shown in figs 1-2.

2) Bin Combination: Using the notation from the bin creation section, if the height of \( B_y \) is equal to the height of a neighbouring bin \( B_n \), forming a continuous flat level, then \( B_y \) is joined to \( B_n \), amalgamating them into one bin. This process is shown in fig 4.

3) Raising Narrowest Bin: If \( B_z \) is too narrow for any remaining piece to fit in, then \text{Height}(B_z) is increased to the level of its nearest neighbour, and then the bin combination process occurs between the two bins. This is only invoked if \( B_z \) is between two higher bins. The space is wasted space, and so can be discarded for the purposes of continuing to construct a solution. If \( B_z \) is not between two higher bins, then the space cannot yet be counted as wasted space. This process is shown in figs 2-3.

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

B. How the Heuristic Decides Where to Put a Piece

This section is a general explanation of the process by which the heuristic decides which piece to pack next and where to put it. This is also summarised in the pseudocode of figure 5, along with the parts of the algorithm that update the bin structure. Section III-C then goes into more detail on this process, using a specific example.

In each bin, there are two places that a piece can be put; in the bottom-left corner or in the bottom-right corner. There are also two orientations that the piece can adopt in both corners. Given a partial solution, we will refer to a combination of piece, bin, corner and orientation as an ‘allocation’. Therefore, there is a total of four allocations to consider for each piece and bin combination, provided that the piece’s width in each orientation is smaller than the width of the bin. An allocation therefore represents one of the set of choices (of a piece and where to put it) that a heuristic could make at the given decision step. A heuristic in this hyper-heuristic system is a function that rates each allocation. The heuristic is evaluated once for each allocation to obtain a score for each allocation.

The heuristic scores an allocation by taking into account a number of features, which are represented as the GP terminals shown in the lower six rows of table I. There are two terminal values describing the piece width and height in its given orientation, two describing the bin, and two describing the bins either side of the allocation bin.

‘Neighbouring’ and ‘opposite’ bins are referred to in table I. A given allocation is either in the lower-left or lower-right corner of the bin, if the allocation is in the lower-left of the bin, the bin to the left of it is referred to as the...
WHILE pieces exist to be packed
  IF narrowest bin is too narrow for any piece AND
      narrowest bin is between two higher bins
    THEN raise narrowest bin to level of nearest neighbour
  ELSE IF at least one piece can fit in a bin
    FOR each allocation
      Save best allocation according to heuristic
    END FOR
    perform the best allocation on the solution
  ELSE
    raise lowest bin
  END IF
END WHILE
TABLE I
THE FUNCTIONS AND TERMINALS AND DESCRIPTIONS OF THE VALUES THEY RETURN

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

‘neighbouring’ bin and the bin to the right is the ‘opposite’ bin. Those labels are reversed when the allocation is in the lower-right of the bin.

For each possible allocation, the values of the terminals are calculated, and the heuristic is evaluated. The allocation for which the heuristic returns the highest value is deemed to be the best, and therefore that allocation is performed on the solution at the current step. In other words, the piece from the allocation is put in the bin from the allocation, in the corner and orientation dictated by the allocation. This process is shown in the example given in section III-C.

C. A packing example

This section works through an example of how the heuristic chooses a piece from those which remain to be packed, and where to put it in the partial solution. It goes into further detail than section III-B. The heuristic we will use in this example is shown in figure 6. It consists of nodes from the GP function and terminal set shown in table I. It contains at least one of each of the functions and terminals, however this is not a requirement. A heuristic in the population could contain any subset of the nodes available.

We will use the heuristic shown in figure 6 to choose a piece from those which remain to be packed (shown in figure 7) and choose where to place it in the partial solution shown in figure 8. The partial solution shows that two pieces have already been packed by the heuristic, one to the left and one to the right. We do not show this process, because it is the same as the one we will explain, and the example will be more descriptive if we show the process in the middle of the packing. There are three bins in the partial solution, labelled one to three, which are defined by the pieces already packed.

The algorithm takes each piece in turn, and evaluates the tree for every possible allocation of that piece. So, first we will consider piece one from figure 7. A piece can be placed in the left or right of a bin, in either orientation, as long as it does not exceed the width of the bin. So, piece one cannot be placed into bin one, because that bin is only 15 units wide, and the piece is 20 units wide in its narrowest orientation. Piece one can be placed into bin three, but only in one orientation, the piece will be too wide if it is placed in its horizontal orientation. Bin two is wide enough to accommodate the piece in either of its orientations.
Figure 9 shows these six valid allocations for piece one in the partial solution, labelled A to F. Each of these six allocations will receive a score, obtained by evaluating the tree once for each allocation. The tree will give a different score for each allocation because the GP terminal nodes will evaluate to different values depending on the features of the allocation.

This process of evaluating the tree is explained here, by taking the examples of allocations A and C from figure 9. Figure 10 shows allocation A in detail. To evaluate the tree for allocation A, we will first determine the values of the terminal nodes of the tree. The ‘width’ (W) and the ‘height’ (H) terminals will take the values 20 and 50 respectively. The ‘bin width left’ (BWL) terminal will evaluate to 35 because that is the horizontal space left in the bin after the piece is put in. The ‘bin height’ (BH) terminal evaluates to zero, because the base of bin two is at the foot of the sheet. The opposite bin for the allocation is bin three, as it is to the right of bin two, and the allocation is in the bottom left corner of bin two. The ‘opposite bin’ (OB) terminal will evaluate to –45, because the opposite bin has a height of 45, and OB is the height of the current bin minus the height of the opposite bin. The neighbouring bin is bin one, because it is to the left of bin two, and the allocation is also in the left of bin two. The ‘neighbouring bin’ (NB) terminal will evaluate to –80, because the neighbouring bin has a height of 80.

Expression 1 shows the tree written in linear form. If we substitute the terminal values into the expression, we get expression 2. This simplifies to expression 3, which evaluates to –44.454 to three decimal places. This value is the score for the allocation of piece one in bin two, in a vertical orientation and on the left side of the bin.

\[
expression 1 = \left( \frac{W \times H}{NB - (BWL \times H)} \right) - (BH + OB) 
\]

\[
expression 2 = \left( \frac{20 \times 50}{-80 - (35 \times 50)} \right) - (0 - 45) 
\]

\[
expression 3 = \left( \frac{1000}{1830} \right) + 45 
\]

Figure 11 shows allocation C in detail. Again, we will calculate the values of the terminal nodes for this allocation in order to evaluate the tree. W and H are now 50 and 20 respectively, they are different from their values in allocation A because the piece is now in the horizontal orientation. BWL evaluates to five, as this is the horizontal space left in the bin after the piece has been placed, shown in figure 11. BH evaluates to zero, as before, because the allocation concerns the same bin. OB and NB evaluate to –80 and –45 respectively, as the allocation is for the bottom right side of the bin rather than the left.

When the terminal values have been substituted in, the tree simplifies to expression 4, which evaluates to 73.103 to three decimal places. This is the score for the allocation of piece one in bin two, in a horizontal orientation and on the right
Table II

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>64, 128 and 256</td>
</tr>
<tr>
<td>Maximum generations</td>
<td>50</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.85</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.1</td>
</tr>
<tr>
<td>Reproduction probability</td>
<td>0.05</td>
</tr>
<tr>
<td>Tree initialisation method</td>
<td>Ramped half-and-half</td>
</tr>
<tr>
<td>Selection method</td>
<td>Tournament selection, size 7</td>
</tr>
</tbody>
</table>

Of these two allocations we have shown, the second allocation has been rated as better by the heuristic because it received a higher score. The other four allocations for this piece are scored in the same way. Then the allocations possible for piece two are scored, of which there are essentially two, shown in figure 12. There are in fact four allocations which are scored for this piece, but it has identical width and height so both orientations will produce the same result from the heuristic. The rest of the pieces that remain to be packed have all of their allocations scored in the same way. Finally, the allocation which received the highest score from the heuristic is actually performed. In other words the piece from the allocation is committed to the partial solution in the orientation and position in the bin dictated by the allocation. Then the bin structure is updated according to the rules explained in section III-A because a new piece has been put into a bin.

For example, the allocation involving piece one in position ‘C’ from figure 9 received a score of 73,103. If no subsequent allocation (involving the same piece or any other piece) received a higher score than this, then this will be the allocation that is performed. Assuming there are no further pieces to be packed that have a width smaller than five, the bins will then be reconfigured as shown in figure 13, because of the process of raising the narrowest bin, shown in section III-A and figures 2 and 3.

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

D. How the Heuristic is Evolved

The hyper-heuristic GP system creates a random initial population of heuristics from the function and terminal set. The individual’s fitness is the height of the solution that it creates when the algorithm in fig 5 is run. Table II shows the GP initialisation parameters. During the tournament selection, if two heuristics obtain the same height, and therefore have the same fitness, the winner will be the heuristic which results in the least waste between the pieces in the solution, not counting the free space at the top of the sheet. Therefore, there is selection pressure on the individuals to produce solutions where the pieces sit next to each other neatly without any gaps. The individuals are manipulated using the parameters shown in table II. The mutation operator is point mutation, using the ‘grow’ method explained in [7], with a minimum and maximum depth of 5, and the crossover operator produces two new individuals with a maximum depth of 17.

We perform three runs for each problem instance, resulting in three heuristics. For each of the runs we vary the population size, as seen in table II in order to investigate the impact that the population size makes on the quality of the resulting heuristic.

IV. Benchmark Problems

We use 46 benchmark instances from the literature to test our hyper-heuristic methodology. The instances used
TABLE III
THE BENCHMARK INSTANCES USED IN THIS PAPER

<table>
<thead>
<tr>
<th>Instance set name</th>
<th>Number of Instances</th>
<th>Number of Rectangles</th>
<th>Sheet Width</th>
<th>Optimal Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valenzuela and Wang (2001)</td>
<td>12</td>
<td>25-1000</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Burke, Kendall and Whitwell (2006)</td>
<td>12</td>
<td>10-500</td>
<td>40-100</td>
<td>40-300</td>
</tr>
<tr>
<td>Ramesh Babu and Ramesh Babu (1999)</td>
<td>1</td>
<td>50</td>
<td>1000</td>
<td>375</td>
</tr>
</tbody>
</table>

are summarised in table III. All of the instances were created from known optimal packings. The Hopper and Turton dataset contains 7 classes of 3 problems each, and each class was constructed from a different sized initial rectangle and contains a different number of pieces. All pieces have a maximum aspect ratio of 7. Valenzuela and Wang created two classes of problem, referred to as ‘nice’ and ‘path’. The nice dataset contains pieces of similar size, and the path dataset contains pieces that have very different dimensions. The dataset from Burke, Kendall and Whitwell contains 12 instances with increasing numbers of rectangles in each. These datasets can be found at http://dip.sun.ac.za˜vuuren/repositories/levelpaper/spp[1].htm. We also use an instance created by Ramesh Babu and Ramesh Babu, containing 50 rectangles of all similar size. The dimensions of the pieces in this instance are given in [38].

The Valenzuela and Wang dataset uses floating points to represent the dimensions of the rectangles. Our implementation uses integers, so to obtain a dataset we can use, we multiplied the data by 100 and rounded to the nearest integer, the results are then divided by 100 so they can be compared to the other results in the literature. For each instance modified in this way, the total area of all the shapes divided by the width of the sheet is never less than 100, so we can say that this procedure never results in an easier instance, and it is fair to compare the results with others in the literature.

V. RESULTS AND DISCUSSION
This section is arranged into three subsections. Section V-A explains the benchmark algorithms with which we compare the evolved heuristics. Section V-B presents the results of comparisons with known metaheuristic and constructive heuristic approaches. Finally, section V-C presents the results of a comparison of our evolved heuristics with a complex human-designed heuristic with constructive and iterative phases.

A. Benchmarks

Table IV shows the overall results we have obtained for the system, with a different random seed at the beginning of every run. We compare our results to two metaheuristic methods described in section II-C3, a genetic algorithm with bottom-left-fill decoder, and a simulated annealing approach with bottom-left-fill decoder. Table IV shows only the best result of the two on each instance. These two metaheuristic approaches are also described in [5], and the results evaluated using a density measure rather than the length of sheet measure used in this paper. To obtain the results that we compare with here, the metaheuristic methods were implemented again in [33].

The ‘best-fit’ style algorithm described in [33], which has achieved superior results to BL and BLF, is used as a constructive, non-metaheuristic, benchmark, and we also compare with the reactive GRASP presented in [44]. The GRASP method does not allow piece rotations, while they are allowed for the heuristics evolved here. The GRASP results would probably not be worse if rotations were allowed, so while we are aware of the difference, we believe the comparison with GRASP is still valuable, as it is a comparison with a complex human designed heuristic with many parameters.

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

B. Comparison with Metaheuristics and Best-Fit

Both of the metaheuristics were unable to find a solution to the NiceP6 and PathP6 problems due to excessive time requirements. In table IV, we highlight, in bold type, the results where our system finds a heuristic which equals or beats the metaheuristic approaches and the best-fit style algorithm from [33] for that instance.

The table shows that in the vast majority of cases, the GP system finds a heuristic in at least one of the three runs that obtains packings better than or equal to the two established metaheuristics and the best-fit algorithm. However, there are four exceptions. For the instance c1p1, the GP system cannot find a heuristic that performs better than either the GA+BLF or the SA+BLF, both of which find the optimal solution at a height of 20. It is known in the literature that the metaheuristic approaches perform well on problems of small size because the search space is small. Also, no heuristic is found that is better than the best-fit heuristic on the three most difficult ‘Path’ problems from [40].

We conclude that the heuristics designed automatically by the GP perform as well, and in many cases better than,
these recent human designed heuristic and metaheuristic approaches.

C. Comparison with the reactive GRASP

The evolved heuristics’ results are competitive with the reactive GRASP method explained in section II-C3, and on the majority of problems, the reactive GRASP obtains results which are the same or better than the evolved heuristics. Table V presents the mean and standard deviation of the differences between the reactive GRASP results and the best evolved heuristic for each problem. The table shows the mean and standard deviation of the differences both as percentages and actual units, and it shows these calculations on all the results and on each individual benchmark set. Table V

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Pieces</th>
<th>Optimal Height</th>
<th>Reactive Heuristic</th>
<th>Evolved Heuristic 64</th>
<th>Evolved Heuristic 128</th>
<th>Evolved Heuristic 256</th>
<th>Reactive GRASP</th>
<th>Max Time to Evolve (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>10</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0.141</td>
</tr>
<tr>
<td>N2</td>
<td>20</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>50</td>
</tr>
<tr>
<td>N3</td>
<td>30</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>N4</td>
<td>40</td>
<td>80</td>
<td>83</td>
<td>83</td>
<td>81</td>
<td>82</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>N5</td>
<td>50</td>
<td>100</td>
<td>106</td>
<td>103</td>
<td>103</td>
<td>105</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>N6</td>
<td>60</td>
<td>100</td>
<td>103</td>
<td>102</td>
<td>102</td>
<td>102</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>N7</td>
<td>70</td>
<td>100</td>
<td>106</td>
<td>102</td>
<td>102</td>
<td>102</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>N8</td>
<td>80</td>
<td>80</td>
<td>85</td>
<td>82</td>
<td>83</td>
<td>82</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>N9</td>
<td>100</td>
<td>150</td>
<td>155</td>
<td>154</td>
<td>151</td>
<td>152</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>N10</td>
<td>200</td>
<td>150</td>
<td>154</td>
<td>152</td>
<td>152</td>
<td>152</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>N11</td>
<td>300</td>
<td>150</td>
<td>155</td>
<td>152</td>
<td>152</td>
<td>152</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>N12</td>
<td>500</td>
<td>300</td>
<td>312</td>
<td>305</td>
<td>311</td>
<td>304</td>
<td>303</td>
<td>303</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Difference in Performance of the Best Evolved Heuristics for Each Instance and the Reactive GRASP, the Figures Represent How Much Greater the Evolved Heuristics’ Results Are</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>Evolved Heuristic 64: 106.73 - 107.03 = -0.30</td>
</tr>
<tr>
<td>Units</td>
<td>Percentage: 1.41%</td>
</tr>
</tbody>
</table>

Table IV presents the mean and standard deviation of the differences between the best evolved heuristics and the recent human designed heuristic and metaheuristic approaches. Table V shows how much greater the evolved heuristics’ results are compared to the reactive GRASP.
Fig. 14. Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 64

Fig. 15. Packing of instance c5p3 to a height of 91. Packed by the heuristic evolved with population 256

Fig. 16. Heuristic that packs fig 14

Fig. 17. Heuristic that packs fig 15

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

D. Example packings

Figures 14 and 15 show two packings achieved for the problem instance c5p3, both of which have a height of 91, just 1 unit over the optimal packing. Figures 16 and 17 show the two evolved heuristics that created these packings, where W and H are the piece width and piece height respectively. OBH and NBH are the opposite bin height and the neighbouring bin height. BWL is the bin width left, and BH is the bin height. All of these GP terminals are summarised in table I. The heuristics have a tree structure like that of figure 6, but could not be displayed in this way because of space constraints. These two heuristics are not easily understood and likely contain redundancy as is normal in the output of a GP run. They are offered not as an explanation of why the pieces were packed in this way, but as examples of the output of our hyper-heuristic. They illustrate that the output heuristics are complex, and take a different form to that which a human programmer would write.
Figure 16 was evolved when the population was set to 64, and figure 17 was evolved when the population was set to 256. Figure 16 is a heuristic that packs the pieces of this instance leaving no gaps at all, as figure 14 shows. Interestingly, figure 17 is a heuristic that decides to pack one of the smallest pieces near to the beginning, seen in the bottom left corner of figure 15. Even though this may seem counter-intuitive to humans, it is a strategy that pays off in the long run for this problem instance.

VI. CONCLUSIONS

This paper has shown that an evolutionary hyper-heuristic approach can generate good quality disposable heuristics for the 2D strip packing problem. For a given instance, the hyper-heuristic evolves a good heuristic for deciding where to place each piece in the current solution. The results obtained by the evolved heuristics were mostly better than both established human designed metaheuristics and a recently published heuristic approach shown to be superior to the well documented heuristic approaches of BL and BLF. In the cases where the results were not better, the evolved heuristics resulted in at most only 5% more waste.

To automatically generate heuristics which are competitive with these state of the art approaches would be a major scientific contribution. However, the constructive heuristics evolved here are better than the human designed ‘best-fit’ algorithm, the state of the art constructive heuristic for this problem. They are also better than the state of the art metaheuristic approaches circa 2001, presented in [5].

The genetic programming hyper-heuristic operates at a higher level of abstraction to previous metaheuristic approaches, by operating on a space of heuristics and not directly on a space of solutions. This means the hyper-heuristic can automatically generate a heuristic which performs well on a given problem instance, allowing good results to be obtained on a wide variety of instances without changing any parameters of the system. All that is required is to input a different problem instance.

VII. FUTURE WORK

Evolutionary algorithms which evolve candidate solutions to a problem instance will output a solution only applicable to that instance. In contrast, the heuristics evolved by the hyper-heuristic methodology shown here can be applied to other problem instances. They will produce a legal solution on an unseen instance because they are methods to solve a problem, they are not solutions. Therefore, to extend this work, we wish to thoroughly test the evolved heuristics on new problem instances, as opposed to treating them as ‘disposable’ heuristics only applicable to the problem instance they are evolved for. We have carried out similar work on evolving re-usable heuristics for the one dimensional bin packing problem in [45], and we will extend this work to see if this is possible in the two dimensional case.

We will apply the evolved heuristics to new instances of varying similarity to the training instance, and will investigate whether a heuristic evolved on one type of problem still performs well on new problems with similar piece sizes. We will also investigate how large the training set needs to be in order for the heuristic to successfully generalise. It is important to note that once evolved, the heuristics can be applied very quickly to an instance because they are constructive, and involve no iteration. Extending the work in this way would potentially justify the long run times in the evolution phase, because the heuristics could then be applied to unseen instances very quickly.

ACKNOWLEDGMENT

This work was supported by ESPRC grant reference EP/C52385/1.

REFERENCES
