Burke, Edmund K. and Hyde, Matthew and Kendall, Graham and Woodward, John (2010) A genetic programming hyper-heuristic approach for evolving 2-D strip packing heuristics. IEEE Transactions on Evolutionary Computation, 14 (6). pp. 942-958. ISSN 1089-778X

A Genetic Programming Hyper-Heuristic Approach for Evolving 2-D Strip Packing Heuristics

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

***Abstract*—We present a genetic programming (GP) system to evolve reusable heuristics for the 2-D 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. This paper contributes to a growing research area that represents a paradigm shift in search methodologies. Instead of using evolutionary computation to search a space of solutions, we employ it to search a space of *heuristics* for the problem. A key motivation is to investigate methods to automate the heuristic design process. It has been stated in the literature that humans are very good at identifying good building blocks for solution methods. However, the task of intelligently searching through all of the potential combinations of these components is better suited to a computer. With such tools at their disposal, heuristic designers are then free to commit more of their time to the creative process of determining good components, while the computer takes on some of the design process by intelligently combining these components. This paper shows that a GP hyper-heuristic can be employed to automatically generate human competitive heuristics in a very-well studied problem domain.**

***Index Terms*—2-D stock cutting, genetic programming, hyper-heuristics.**

I. Introduction

HYPER-HEURISTICS can be thought of as heuristics which search a space of heuristics, as opposed to searching a space of solutions directly [1] (which, of course, is the conventional approach to employing evolutionary algorithms). We employ genetic programming (GP) 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 set of automatically designed heuristics that can be reused on new problems, and which are competitive with the best human designed constructive heuristic.

This paper presents such a hyper-heuristic system for the 2-D strip packing problem, where a number of rectangles must be placed onto a sheet with the objective of minimizing 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 (non-deterministic polynomial-time) hard [2], 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 minimizing waste.

Indeed, many industrial problems are not limited to just rectangles (for example textiles, leather, etc.) and this presents another challenging problem [3]. 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 [4]. 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 paper, the rectangular pieces are free to rotate by 90°, 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 nonguillotine version of the problem, which has no such constraint on how the pieces are cut.

The motivation behind this paper is to develop a hyper-heuristic GP methodology which can automatically generate a novel heuristic for any class of 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, work employing such systems for other problem domains has been published (see Section III-B).

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 [5], [6]. 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. As we will show, the heuristics we evolve 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.

The outline of this paper is as follows. In Section II, we discuss some of the motivations and philosophy behind this line of research. In Section III, we introduce the background literature on GP, hyper-heuristics, and 2-D stock cutting approaches. Section IV presents our algorithm. Section V describes the benchmark problem instances used in this paper, and Section VI presents the results of the evolved heuristics on new problem instances not used during their evolution. The results of the best evolved heuristic are analyzed and compared against recent results in the literature on benchmark instances. Finally, conclusions and ideas for future work are given in Sections VII and VIII.

II. Motivation for This Research Area

The "No Free Lunch" theorem [7], [8] shows that all search algorithms have the same average performance over all possible discrete functions. This would suggest that it is not possible to develop a general search methodology for all optimization problems as, over all possible discrete functions, "no heuristic search algorithm is better than random enumeration" [9]. However, it is important to recognize that this theorem is *not* saying that it is not possible to build search methodologies that are *more* general than is currently possible. It is often the case in practice that search algorithms are developed for a specific group of problems, for instance timetabling problems [10], [11]. Often the algorithms are developed for a narrower set of problems within that group, for instance university course timetabling [12] or exam timetabling problems [13]. Indeed, algorithms can be specialized further by developing them for a specific organization, whose timetabling problem may have a structure very different to that of another organization with different resources and constraints [14]. At each of these levels, the use of domain knowledge can allow the algorithms to exploit the structure of the set of problems in question. This information can be used to intelligently guide a heuristic search.

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

The subject of this paper is a hyper-heuristic system which automatically designs heuristics, using a GP algorithm. The heuristics are automatically designed by using GP to intelligently combine a set of human defined components. While the specification of the components themselves is not automated, the methodology as a whole requires less human input than would be required to manually design fully functional heuristics. Fukunaga states that humans can readily identify good potential components of methods to solve problems, but that the task of combining them could benefit from automation [16]. As problems in the real world become more complex, identifying ways to automate this process may become fundamental to the design of heuristics, because it will become more difficult to manually combine their potential components in ways that fully exploit the structure of a complex problem.

There are a number of advantages of developing a methodology to automatically design heuristics. There is a possibility of discovering new heuristics that are unlikely to be invented by a human analyst, due to their counterintuitive nature. Another advantage is that a different heuristic can be created for each subset of instances, meaning that the results obtained on each are more likely to be better than those obtained by one general heuristic. Human created heuristics are designed to perform well over many problem instances, because it would be too time consuming to manually develop a new heuristic specialized to every subset of instances. A hyper-heuristic approach, such as the one described in this paper, can specialize heuristics to a given problem class, at no extra human cost. The evolutionary algorithm need only be run again to produce a new heuristic.

One of the long term goals of this research direction is in making optimization tools and decision support systems available to organizations who currently solve their problems by hand, without the aid of computers. Examples of such organizations could be, for example, a primary school with a timetabling problem, or a small delivery company with a routing problem. It is often prohibitively expensive for them to employ a team of analysts to build a bespoke heuristic, which would be specialized to their organization's problem. A more general system that automatically creates heuristics would be applicable to a range of organizations, potentially lowering the cost to each. It may be that there is a trade-off between the generality of such a system, and the quality of the solutions it obtains. However, organizations for whom it is too expensive to commission a bespoke decision support system, are often not interested in how close their solutions are to optimal. They are simply interested in how much better the solutions are than those they currently obtain by hand. For example, consider a small organization that currently solves its delivery scheduling problem by hand. This organization may find that the cost of commissioning a team of humans to design a heuristic decision support system, would be far greater than the benefit the company would get in terms of better scheduling solutions. However, the cost of purchasing an "off the shelf" decision support system which can *automatically* design appropriate heuristics, may be lower than the resulting reduction in costs to the organization. If the solutions are good enough, and they are cheap enough, then it begins to make economic sense for more organizations to take advantage of heuristic search methodologies. Hyper-heuristic research aims to address the needs of organizations interested in *"good enough, soon enough, cheap enough"* [17] solutions

to their optimization problems. Note that "good enough" often means solutions better than currently obtained by hand, "soon enough" typically means solutions delivered at least as quick as those obtained by hand, and "cheap enough" usually means the cost of the system is low enough that its solutions add value to the organization.

III. Background

A. Genetic Programming

GP (see [18]–[20]) 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 IV-E.

B. Hyper-Heuristics

Hyper-heuristics are defined as heuristics that search a space of heuristics, as opposed to searching a space of solutions directly [1]. Research in this area is motivated by the goal of raising the level of generality at which optimization systems can operate [17]. 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 [21] and a learning classifier system [22] have been used as hyper-heuristics for the 1-D bin packing problem. A genetic algorithm with an adaptive length chromosome and a tabu list was used in [23] as a hyper-heuristic. A case based reasoning hyper-heuristic is used in [24] for both exam timetabling and course timetabling. Simulated annealing is employed as a hyper-heuristic in [25] for the shipper rationalization problem. A tabu search hyper-heuristic [26] 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 [27]. Three new hyper-heuristic architectures are presented in [28], 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 [29]. A distributed choice function hyper-heuristic is presented in [30].

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. Examples of other work in this growing research area are as follows. The "CLASS" system presented in [16], [31], and [32] is an automatic generator of local search heuristics for the satisfiability (SAT) problem, and is competitive with human-designed heuristics. A different methodology for SAT is presented in [33], where heuristics are more parsimonious and faster to execute. 1-D bin packing heuristics, evolved in [34]–[36], have superior performance to the human-designed best-fit heuristic, even on new instances much larger than those in the training set. A hyper-heuristic approach has also been applied to the traveling salesman problem [37], and to evolve dispatching rules for the job shop problem [38].

C. 2-D Stock Cutting Approaches

1) *Exact Methods:* Gilmore and Gomory [39] first used a linear programming approach in 1965. Tree search procedures have been employed to produce optimal solutions for small instances of the 2-D guillotine stock cutting problem [40] and 2-D nonguillotine stock cutting problem [41]. The method used in [40] has since been improved in [42] and [43]. Recent exact approaches can be found in [44]–[47]. It is recognized that, in general, these methods do not provide good results on large instances, due to the problem being NP-hard.

2) *Heuristic Methods:* Baker *et al.* define a class of packing algorithms named "bottom up, left justified" (BL) [48]. These algorithms maintain bottom left stability during the construction of the solution. This means that every piece cannot move further downward or to the left. The heuristic presented in [48] 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 justifies it. While this heuristic is intuitively simple, implementations are often not efficient because of the difficulty in analyzing the holes in the solution for the points where a piece can be placed [49]. Chazelle presents an optimal method for determining the ordered list of points that a piece can be put into, using a "spring" representation to analyze the structure of the holes [49].

These heuristics take, as input, a list of pieces, and the results rely heavily on the pieces being in a "good" order [48]. Theoretical work presented by Baker *et al.* [48] and Brown *et al.* [50] show the lower bounds for heuristic 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 bottom-left (BL) or BLF results in performance increases of between 5% and 10%.

Zhang *et al.* [51] use a recursive algorithm, running in $O(n^3)$ time, to create good strip packing solutions, based on the "divide and conquer" principle. Two heuristics for the strip cutting problem with sequencing constraint are presented by Rinaldi and Franz [52], based on a mixed integer linear programming formulation of the problem.

A "best-fit" style heuristic was presented in [53]. This algorithm is shown to produce better results than previously published heuristic algorithms on benchmark instances [53]. The details of this heuristic are given in Section IV-A. This heuristic is hybridized with simulated annealing in [54]. The methodology involves using best-fit to pack most of the pieces, and then using simulated annealing to iteratively reorder the remaining pieces, and repack them with bottom-left-fill. This approach obtains significantly better results than previously published methodologies, on almost all of the

benchmark problems. We use the nonhybridized version of best-fit for comparison in this paper, because our evolved heuristics are contructive, and best-fit is the best human created constructive heuristic in the literature.

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 [55] 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 [48] and [56] 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 [57] use a genetic algorithm in the same way as Jakobs, to evolve an order of pieces, but they use a slightly different heuristic to pack the pieces, and different genetic algorithm parameters, improving on Jakobs' results.

Valenzuela and Wang [58] 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 GP trees in this paper, which represent heuristics. However, the slicing trees are not heuristics. 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 [48]. 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 [59] and Faina [60] both use a simulated annealing approach in this way, and achieve good results on problems of small size. Also, Bortfeldt [61] uses a GA which operates directly on the representations of strip packing solutions.

A reactive greedy randomized adaptive search procedure (GRASP) is presented in [62] for the 2-D 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: 1) one of four methods of selecting the next piece to pack is chosen; 2) a method of randomizing the piece selection is chosen from a choice of four; 3) there are five options for choosing a parameter δ, which is used in the randomization method; and 4) 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 [56]. Their sequential value correction (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 [56].

We compare with the results of the reactive GRASP in Section VI, 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 in certain respects from the aims of other work in the literature.

The aim of the vast majority of the literature is to generate good quality solutions. The aim of this paper is to focus on a methodology capable of generating good quality *heuristics*. The quality of the results obtained by the automatically designed heuristics is of high importance, but we do not aim only for better results than the state of the art hand crafted 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 2-D strip packing [56], [62], which use a constructive phase *and* an iterative phase. We also show that the automatically generated constructive heuristics can obtain better results than the human designed state of the art constructive heuristic, presented in [53].

IV. Methodology

In Section IV-A, we explain the functionality of the best-fit heuristic from [53], [54], to which we compare our evolved heuristics, and which provides the inspiration for our packing framework. In Section IV-B, we explain the representation of the problem that we use, and how it is updated each time a piece is placed into the solution. Section IV-C 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 IV-D to further clarify this process. Section IV-E explains how the heuristics themselves are evolved, detailing the GP parameters and the method by which the heuristics are trained.

A. Best-Fit Heuristic

The best-fit heuristic [53] is explained here because it provides the inspiration for the framework described in Section IV-C and Fig. 5. We also compare our evolved heuristics to this heuristic in Section VI.

The heuristic returns the result of three separate attempts at packing, once with each of three placement policies. For each policy, the pieces are packed one at a time, each into the current lowest available slot on the sheet. The pieces are free to be rotated by 90°, and the piece chosen to be packed is the one which fills the most of the width of the slot. Note that the piece can be placed in the left or right sides of the slot, and it is the current placement policy that determines which side the piece is placed. The first placement policy is to always put the piece into the lower left corner of the slot. The second policy is to put the piece next to the tallest neighboring piece. Finally, the third policy is to put the piece next to the shortest neighboring piece. The three policies result in different solutions, and the best of the three solutions is returned as the result of the heuristic.

Our evolved heuristics pack the pieces in a similar way to the best-fit heuristic, because all the pieces are considered for packing at each step, not just the first in the sequence given to it. However, in contrast, best-fit only considers one slot for the pieces, whereas the heuristics evolved in this paper consider *all* slots. We also give our evolved heuristics three attempts at packing, once with each placement policy, as is the case with best-fit.

B. Representation of the Problem

Our hyper-heuristic system evolves a constructive heuristic, which considers the strip packing problem to be a sequence of steps, where a piece must be placed at each step. 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 "slots," the number and configuration of which will change at every step. Each slot has a height (the distance from the base of the sheet to the base of the slot), a lateral position, and a width. Each slot represents a position in the solution where a piece can be placed, and the slot structure is refreshed after a piece is placed.

At the beginning of the packing process the sheet will be represented as just one slot, with height zero and width equal to the width of the sheet. As an example, Fig. 1 shows the slot configuration (two slots, with different heights and widths) when one piece has been placed onto the sheet in the lower left corner. The slots are shown by horizontal lines, and the left and right limits of the slot are shown by black squares. The dashed line extending from the highest slot signifies that the width of the slot extends beyond the top of the piece and continues until it reaches the right hand side of the sheet.

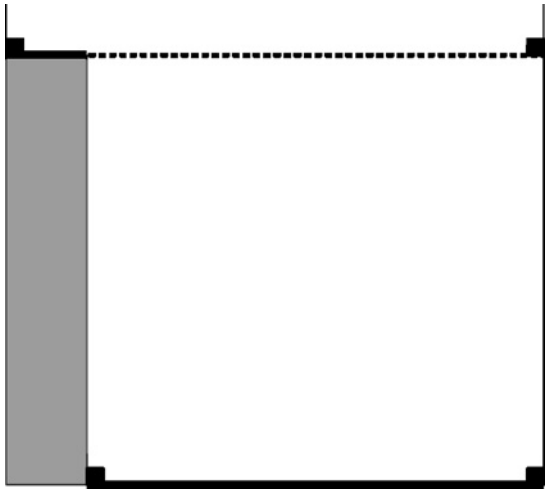

Fig. 1. Two slots after one piece has been packed, the highest slot extends out to the right as far as the edge of the sheet.

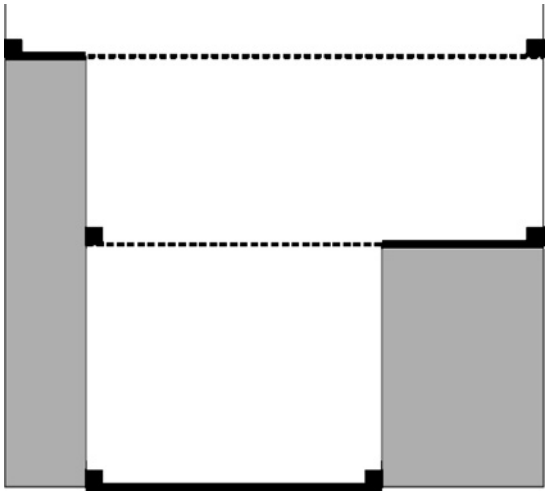

Fig. 2. New slot extends out to the left, as far as the edge of the first piece.

Figs. 1–4 show a step by step example of three more pieces being packed and how the slot structure changes as each piece is packed. Fig. 2 shows the slot structure after a second piece has been placed into the lower right corner. One can observe that the bold black horizontal lines represent the highest horizontal surfaces at any given horizontal position in the packing. The slots are defined by these black lines, by extending their widths in both directions to the nearest vertical edge of a piece or the edge of the sheet (this is shown by the dashed lines). There are now three slots in the partial solution.

Fig. 3 shows the slot structure after a piece has been placed into the lower left corner of the lowest slot. There are now four slots, and one can see that the widths of the slots extend as far as the nearest vertical edge in both directions. After the fourth piece has been placed into the second highest slot, Fig. 4 shows the slot structure. There are now two slots, as the height of the fourth piece matches that of its neighbor to the right. As the fourth piece hangs over the right edge of the piece below it, the narrowest slot from Fig. 3 is not generated this time, as the surface is no longer the highest at this horizontal position.

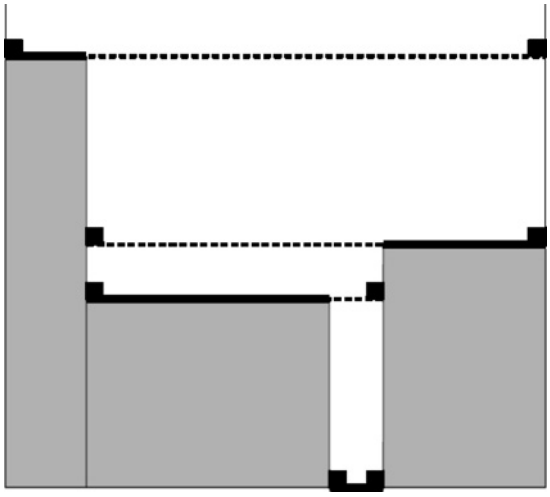

Fig. 3. There are now four slots after the third piece is placed. The black squares show the limits of the slot width.

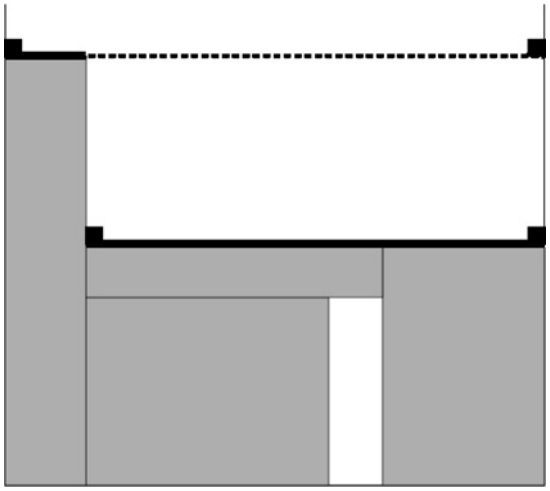

Fig. 4. When the fourth piece is placed, two slots remain because the height of the piece matches that of the piece to its right.

The slot structure is refreshed after each piece is placed, and the process is analogous to pointing a laser vertically downward onto the solution and sweeping it from left to right. All of the surfaces hit by the laser become the bold black horizontal lines, and from these the slots are defined, by extending them left and right to the nearest vertical edges.

C. 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 summarized in the pseudocode of Fig. 5. Section IV-D then goes into more detail on this process, using a specific example.

A piece can adopt two orientations in a slot. Given a partial solution, we will refer to a combination of piece, slot, and orientation as an "allocation." Therefore, there are two allocations to consider for each piece and slot combination, provided that the piece's width in each orientation is smaller than the width of the slot. An allocation therefore represents one of the set of choices (of a piece and where to put it) that

```
FOR each of three placement policies
   WHILE pieces exist to be packed
      IF at least one piece can fit in any slot
         FOR each allocation
            evaluate heuristic on allocation
            obtain a score from the evaluation
            save highest scoring allocation
         END FOR
         perform the best allocation on the solution
      END IF
      update slot structure
   END WHILE
END FOR
RETURN best solution from the three placement policies
```

Fig. 5. Pseudocode showing the overall program structure within which a heuristic operates. Packing policies are explained in Section IV-A.

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 its features, which are represented as the GP terminals shown in Table I. There are three terminal values describing the piece width (W), height (H), and area (A) in its given orientation. There are two representing the slot width (SW) and the slot height (SH), and one which represents the slot width left (SWL), the remaining horizontal space in the slot if the piece were to be put in. The sheet dimensions are represented by terminals for the sheet width (SHW) and sheet height (SHH). The sheet height is calculated as the height of the optimum solution multiplied by 1.5. Constants are included for the heuristics to use, in the form of ephemeral random constants, detailed in [18].

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 slot from the allocation, in the orientation dictated by the allocation. This process is shown in the example given in Section IV-D.

In a similar way to the human created best-fit heuristic (see Section IV-A), an evolved heuristic obtains a result by returning the best of three complete attempts at packing, one with each of three placement policies. When an allocation has been chosen by the heuristic, the piece can either be placed in the left or right of the chosen slot. The location of the piece is determined by the current packing policy. The first placement policy is to always put the piece into the lower left corner of the slot. The second policy is to put the piece next to the tallest neighboring piece. Finally, the third policy is to put the piece next to the shortest neighboring piece. The three policies result in different solutions, and the best of the three solutions is returned as the result of the heuristic.

D. 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 IV-C. The heuristic we will use in this example is

TABLE I
FUNCTIONS AND TERMINALS, AND DESCRIPTIONS OF THE VALUES THEY RETURN

Name	Label	Description
+	+	Add two inputs
-	-	Subtract second input from first input
*	*	Multiply two inputs
%	%	Protected divide function
Width	*W*	The width of the piece
Height	*H*	The height of the piece
Area	*A*	The area of the piece
Slot Height	SH	Slot height, relative to base of sheet
Slot Width Left	SWL	Difference between the slot and piece widths
Sheet Width	SHW	Width of the sheet
Sheet Height	SHH	Height of optimum solution multiplied by 1.5
Constant	ERC	Ephemeral random constant

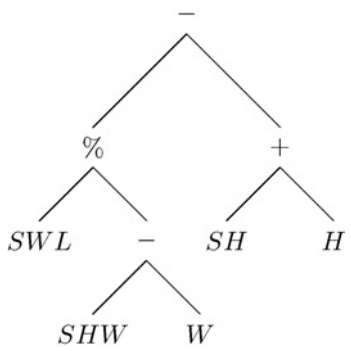

Fig. 6. Example heuristic.

shown in Fig. 6. It consists of nodes from the GP function and terminal set shown in Table I. A heuristic in the population could contain any subset of the nodes available. Recall that the heuristic performs three complete packings, one for each placement policy (described in Section IV-A), and returns the best solution of the three. This example will use the first placement policy, where the piece is always put into the lower left corner of the slot. For the other two placement policies, the same methodology is used, but the rules of the other placement policies will govern whether the piece is placed into the left or right of the slot.

We will use the heuristic shown in Fig. 6 to choose a piece from those which remain to be packed (shown in Fig. 7) and choose where to place it in the partial solution shown previously in Fig. 2. 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 slots in the partial solution, which are defined by the pieces already packed.

For each placement policy, 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 Fig. 7. A piece can be placed into a slot in either orientation, as long

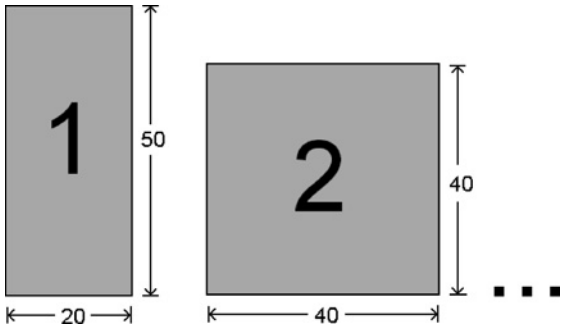

Fig. 7. Pieces we will consider for packing.

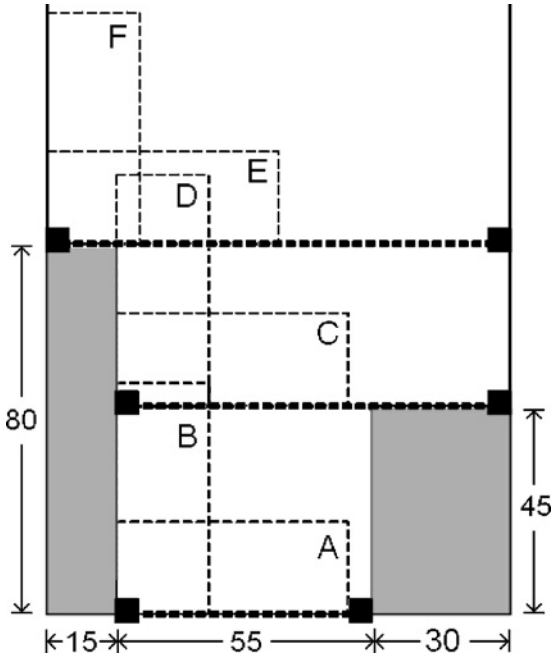

Fig. 8. All of the places where piece one from Fig. 7 can go.

as it does not exceed the width of the slot. Piece one does not exceed the width of any of the slots, so it can be considered for allocation into all three slots. Fig. 8 shows these six valid allocations for piece one in the partial solution, labeled 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. One can see that two of these six allocations represent placing the piece suspended in the middle of the solution, with no piece below it. This is permitted by the representation, because of the possibility of an even wider piece being chosen to be placed across a gap, and we expect that a good evolved heuristic will never choose such an allocation when it can be placed further down in the solution.

The process of evaluating the tree is explained here, by taking the examples of allocations A and B from Fig. 8. Fig. 9 shows allocation A in detail. To evaluate the tree for allocation A, we will first determine the values of the terminal nodes of the tree. The *W* and the *H* terminals will take the values 50 and 20, respectively. The SWL terminal will evaluate to 5

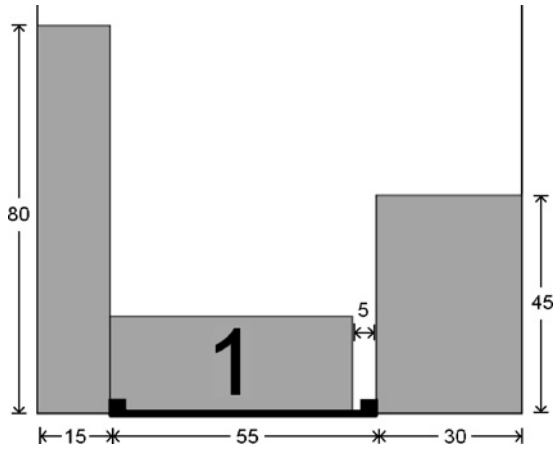

Fig. 9. Allocation A from Fig. 8 in detail.

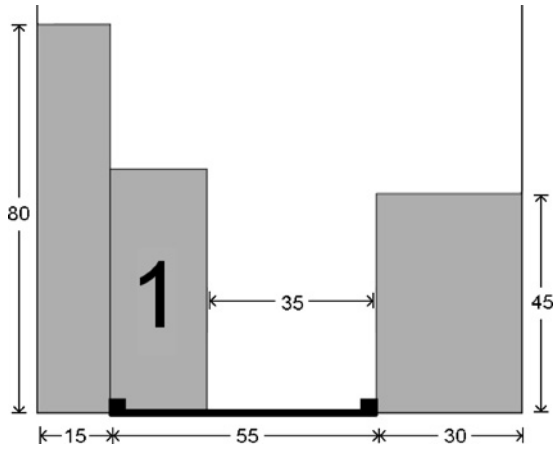

Fig. 10. Allocation B from Fig. 8 in detail.

because that is the horizontal space left in the slot after the piece is put in. The *SH* terminal evaluates to zero, because the base of the slot is at the foot of the sheet. The SHW terminal evaluates to 100, because this is the width of the entire sheet.

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 -19.9. This value is the score for the allocation of piece one in the lowest slot, in a horizontal orientation

$$\left(\frac{\text{SWL}}{\text{SHW} - W}\right) - (\text{SH} + H) \tag{1}$$

$$\left(\frac{5}{100 - 50}\right) - (0 + 20) \tag{2}$$

$$\left(\frac{5}{50}\right) - 20. \tag{3}$$

Fig. 10 shows allocation B 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 20 and 50, respectively, they are different from their values in allocation A because the piece is now in the vertical orientation. SWL evaluates to 35, as this is the horizontal space left in the slot after the piece has been placed, shown in Fig. 10. SH evaluates to zero, as before, because the allocation concerns the same slot. The SHW terminal still evaluates to 100.

When the terminal values have been substituted in, the tree simplifies to expression 4, which evaluates to -49.56 to two decimal places. This is the score for the allocation of piece one in the lowest slot, in a horizontal orientation

$$\left(\frac{35}{80}\right) - 50. \tag{4}$$

Of these two allocations, the first 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 three, shown in Fig. 11. There are, in fact, six allocations that 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 slot dictated by the allocation. Then the slot structure is updated because a new piece has been put into a bin. For example,

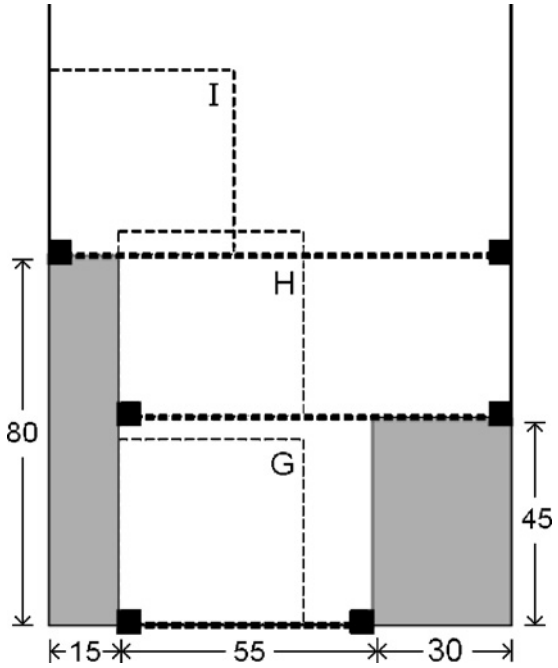

Fig. 11. Three potential allocations for piece two.

TABLE II
INITIALIZATION PARAMETERS OF EACH GENETIC PROGRAMMING RUN

Population Size	1000
Maximum Generations	50
Crossover Probability	0.85
Mutation Probability	0.1
Reproduction Probability	0.05
Tree Initialization Method	Ramped half-and-half
Selection Method	Tournament selection, size 7

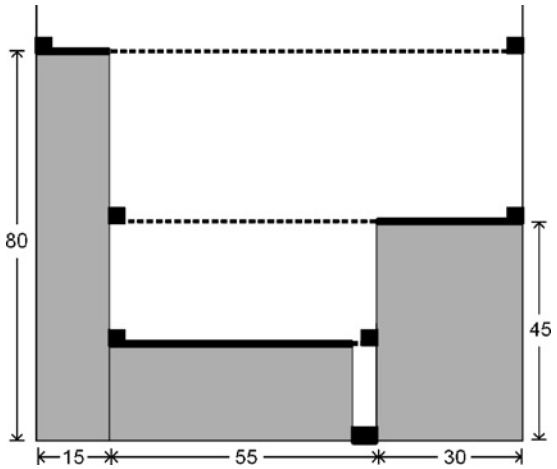

Fig. 12. New partial solution after the allocation that received the highest score has been performed.

the allocation involving piece one in position "A" from Fig. 8 received a score of -19.9. 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. The slots will then be configured as shown in Fig. 12.

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 slot structure that has occurred. When all of the pieces have been packed, the result for the first placement policy is stored, and the process begins again for the second placement policy, starting with an empty sheet again (the details of the placement policies can be found in Section IV-A). The packing process for the second placement policy will be the same as for the first, but the piece could be placed in the right side of a slot if the neighboring piece to the right is larger than the piece to the left.

E. 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 total of the heights of the solutions that it creates when the algorithm in Fig. 5 is run for each instance in the training set. The fitness is to be minimized, because lower and more compact solutions are better. Table II shows the GP initialization parameters. During the tournament selection, if two heuristics obtain the same fitness, and therefore have achieved the same total height on the training instances, the winner will be the heuristic which results in the least waste between the pieces in the solutions, 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 fit next to each other neatly without any gaps. The individuals are manipulated using the parameters shown in Table II. The mutation operator randomly selects a node, internal nodes are selected with 90% probability, and leaf nodes with 10% probability. The subtree that the selected node defines is replaced with a randomly generated subtree using the "grow" method explained in [18], with a minimum and maximum depth of 5. The crossover operator produces two new individuals with a maximum depth of 17.

TABLE III
DETAILS OF THE EIGHT TRAINING INSTANCE CLASSES

Class Name	Number of Pieces	Sheet Width	Optimum Height	Training Instances
N1	10	40	40	10
N2	20	30	50	10
N3	30	30	50	10
N4	40	80	80	10
N5	50	100	100	10
N6	60	50	100	10
N7	70	80	100	10
N8	80	100	80	10

We wish to evolve general heuristics, applicable to more than the instance(s) they are evolved on. To achieve this aim, we use a training set to evolve the heuristics, and then report the results on a separate test set. The instances currently in the literature are varied and numerous enough to compare solution methods, but they are not adequate for the automatic *training or evolution* of solution methods (heuristics). To train a heuristic, one needs a large set of training instances which are similar to each other in some way. We have created such a set using the generation method for the existing benchmark instances referred to as N1–N12 (introduced in [53]). Our aim is to investigate if the evolved heuristics are capable of maintaining their performance on new instances of the same class as those they were evolved on, and on different classes. We only evolve heuristics on the instances from the N1–N8 classes, because of the run times involved in repeatedly packing larger instances during the evolution process. It is also interesting to investigate if the evolved heuristics maintain their performance on instances larger than those they were trained on.

The training instances from the classes N1–N8 were each created with a known optimum solution, because they are generated by iteratively making guillotine cuts across rectangles, starting with a rectangle of the dimensions given in Table III. After the first cut is made, there are two rectangles, and the next cut is made across one of those. Then the next cut is made across one of the three, and so on. We generate ten training instances for each class in this way, so each of the ten instances in a class is generated from the same starting rectangle. The N1–N12 benchmark instances are widely used in the literature, and so we compare the performance of our evolved heuristics with that of other approaches, on these instances. We perform ten runs for each problem instance class, resulting in 80 heuristics.

V. BENCHMARK PROBLEMS

In this paper, we use 46 benchmark instances from the literature to test our evolved heuristics. The instances used are summarized in table IV. All of the instances were created from known optimal packings. The Hopper and Turton dataset contains seven classes of three problems each, and each class was constructed from a different sized initial rectangle and contains a different number of pieces. All pieces have a

TABLE IV
BENCHMARK INSTANCES USED IN THIS PAPER

Instance Set Name	Number of Instances	Number of Rectangles	Sheet Width	Optimal Height
Hopper and Turton (2001)	21	16–197	20–160	20–240
Valenzuela and Wang (2001)	12	25–1000	100	100
Burke *et al.* (2006)	12	10–500	40–100	40–300
Ramesh Babu and Ramesh Babu (1999)	1	50	1000	375

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. We also use an instance created by Ramesh Babu and Ramesh Babu, containing 50 rectangles all of similar size. The dimensions of the pieces in this instance are given in [57].

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 10^6, the results are then divided by 10^6 so they can be compared to the other results in the literature. This procedure never reduces the accuracy of the values, and so it is fair to compare the results with others in the literature.

VI. RESULTS AND DISCUSSION

In this section, we compare our evolved constructive heuristics mainly to the best-fit heuristic, as it is the best human created constructive heuristic in the literature. The best-fit heuristic was proposed in two papers [53], [54] with the latter using an improved representation to handle floating point data without rounding. Due to the implementation differences, the two papers report some variance in the results. In our result tables here, we refer to the two implementations as "version 1" and "version 2."

The results section is divided into four subsections. Section VI-A reports the results of evolved heuristics on new instances of the same class as those they were evolved on. Section VI-B reports the results of those same heuristics on different classes of problem instance. Section VI-C shows the results of heuristics evolved on more than one problem class, which leads to much more general heuristics. The best evolved heuristic is then analyzed in greater detail in Section VI-D.

A. Performance on New Instances of the Same Class

In this section, we report the results of the heuristics when tested on instances from the same class as those they were evolved on. Table V summarizes the results. Each row of the table represents the results of ten heuristics, each evolved on ten instances from the class in the first column. The values are the results on the benchmark instance of that class from the literature. The second and third columns represent the results of the two implementations of best-fit, from [53] and [54]. The fourth shows the result of the best heuristic from the ten which were evolved, and the fifth column shows their average result. Note that the "best" heuristic is the heuristic that obtained the best result on the *training set*, not the test set, so it is valid to say that Table V shows how the heuristics generalize to new instances of the same type.

TABLE V
RESULTS OF THE EVOLVED HEURISTICS WHEN TESTED ON THE BENCHMARK INSTANCE OF THE SAME CLASS AS THOSE ON WHICH IT WAS EVOLVED

Instance	Best-Fit Version 1	Best-Fit Version 2	Best Evolved Heuristic	Average Performance
N1	45	45	44	45.5
N2	53	53	54	54.3
N3	52	52	52	52.7
N4	83	86	83	84.5
N5	105	105	106	105.2
N6	103	102	105	103
N7	107	108	102	104
N8	84	83	83	82.8

Instance is unseen by the heuristic during its evolution, and thus these results represent the ability of the heuristics to generalize to new instances.

The average results in the table show that the evolved heuristics have roughly the same performance, and are sometimes better than best-fit on these benchmark problems. Indeed, the average results for the heuristics evolved on classes N7 and N8 are better than both implementations of best-fit, which means that the GP can successfully evolve heuristics which can beat a human created heuristic on new instances. This shows that heuristics can be evolved with this system to be specialized on a particular class of problem. The next section shows the results of the heuristics when tested on problems of a different class.

B. Performance on New Instances of Different Classes

Table VI shows the results of applying the evolved heuristics to instances of a different class to those they were evolved on. Each row of the table represents the ten heuristics that were evolved on the training set from the class in the first column. The values in a row represent the average results of the ten heuristics on the benchmark test instances named in the top row. One can see from this table that the heuristics are consistent and robust on new instances of the same class as those they were evolved on. The values highlighted in bold represent the results on the benchmark instance from the same class as the heuristics' training set.

The results shown so far indicate that evolving on instances of only one class does not produce general heuristics. The heuristics appear to be specialized to one class of instance, at the expense of their reliability on other classes of instance.

C. Improving Generality by Evolving Heuristics on Three Classes

To investigate if we could increase the level of generality of the evolved heuristics, new heuristics were evolved on instances from three different classes. We evolved heuristics

TABLE VI

RESULTS OF THE EVOLVED HEURISTICS WHEN TESTED ON THE BENCHMARK INSTANCES OF ALL CLASSES

Training Class	Test Instance									
	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
Class N1	**45.5**	55	54.3	91.5	107.5	108.1	117.5	87.8	157.5	158.8
Class N2	49.3	**54.3**	53.1	95.2	118.9	105.4	116	93.1	161.5	158.4
Class N3	44.1	54.5	**52.7**	84.5	109.8	104.5	106.4	83.3	157.2	155.4
Class N4	46.2	54.7	56	**84.5**	105.3	106.2	122	83.6	168.8	157.5
Class N5	49.1	54.6	60.3	87.7	**105.2**	103.2	119.9	82.9	168.8	156.5
Class N6	42.8	54.8	53.2	83.9	106.8	**103**	106.2	84.2	156.6	157
Class N7	41.9	54.6	53.3	83.5	106.7	104	**104**	83	157.7	153.7
Class N8	45.8	56.7	61	84.8	105.2	108.1	118	**82.8**	178.2	160.1

Bold values indicate the results where the heuristics are tested on instances of the same class, and so they match the values in Table V.

on three sets of classes. Five instances were included in the training set from each class, making 15 instances in total. The first set of ten heuristics was evolved on instances of classes N3–N5. The second set was evolved on classes N4–N6, and the third set was evolved on instances from N5–N7. The evolved heuristics were then tested on the N1–N10 benchmark instances from the literature, and compared against the human created best-fit heuristic.

Table VII shows a summary of the results obtained by the heuristics. The first column shows the test instances, and the second and third columns show the results of the two implementations of best-fit. The remaining six columns represent the results of the best heuristics and the average results of the ten heuristics. The "best" heuristic of the ten is defined as the heuristic that achieves the best results on the training set, not on the test set, so the results display the ability of the GP methodology to produce heuristics which can generalize to new instances.

The table shows that the evolved heuristics can now obtain results that are competitive with, and often better than, best-fit across all of the N1–N10 benchmark instances. It is important to emphasize that these heuristics are being reused on new instances of classes different to those they were evolved on, and, in contrast to the heuristics presented in Table VI they maintain their performance on these different instances. They have evolved to be more general because they have seen more than one type of instance during their evolution. It is interesting to note that this level of generality can be evolved by exposing the heuristics to just three different classes. Furthermore, recall that the evolved heuristics do not perform any postprocessing on the solution after it is completed. For example, any pieces which are extending vertically out of the top of the solution are not taken out and replaced horizontally, as is the case with the best-fit heuristic [53]. If postprocessing was perfomed, then in some cases the solutions would be one or two units better. Specific examples of this can be seen in Section VI-D.

D. Analyzing the Best Evolved Heuristic

This section provides some additional results from the best evolved heuristic for the N4+N5+N6 classes. This is the heuristic which performed best on the N1–N10 instances out of the three best evolved heuristics in Table VII. In this section, we test the heuristic on further benchmark datasets from the literature, and compare it against the best-fit heuristic, direct metaheuristic approaches, and a reactive GRASP approach. We then analyze three results from this heuristic in detail using graphical representations of the solutions.

```
(−(−(+ (+ (−(* (+ SH SHH) (+ (+ (−(* (+ SH SHH) (+ (*
(+ SHH (+ SHH H)) (−(+ H W) (* 2 SH))) (+ SHH (+ SHH
H)))) H) (% (+ (* −4.839 SH) (* 4 SHH)) (* SWL (+ SHH
H)))) (+ (+ (* (% SWL SHH) (+ (% SHW 0.963) (% 0.963
SH)))(−(+ H W) (* 3 SH))) (+ SHH H)))) H) (% (+ (−(−H
SH) SH) (+ SHH SHH)) (+ (* (+ SHH (+ SHH H)) (−(+ H
W) (* 2 SH))) (−W (−1 SH))))) (+ (+ (* (% SWL SHH) (−
H SH)) SH) (* (−(+ (+ (−(* (+ SH SHH) (+ (* (+ SHH (+
SHH H)) (−(+ H W) (* 2 SH))) (+ SHH (+ SHH H)))) H)
(% (+ (* −4.839 SH) (* 6 SHH)) (* SWL (+ SHH H)))) (+
(+ (* (% SWL SHH) (+ (% SHW 0.963) (% 0.963 SH))) (+
(−(−H SH) SH) (% SHW 0.963))) (+ SHH H))) (% SWL
SHH)) SHH))) (−H SH)) SHH)
```

Fig. 13. Best evolved heuristic, with some obvious simplifications made.

Fig. 13 shows the evolved heuristic, expressed in prefix notation, with the obvious simplifications made from its raw form. Note that it contains two large repeated sections of code. It also contains many repeated subtrees. For example (+ (* −4.839 SH) (* *X* SHH)) is repeated twice, where *X* is 4 and 6. This expression increases when the slot height is lower, and so could contribute to prioritizing lower slots. Another example that has this property is (+ (% SHW 0.963) (% 0.963 SH)), which occurs twice.

Table VIII shows the results that the best evolved heuristic obtains. We compare its results to two metaheuristic methods described in Section III-C3, a genetic algorithm with bottom-left-fill decoder, and a simulated annealing approach with bottom-left-fill decoder. Table VIII 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 [53]. The results of the best-fit algorithm from [53], which has achieved superior results to BL and BLF, is used as a constructive heuristic benchmark, and we also compare with the reactive GRASP presented in [62]. The GRASP method does not allow piece rotations, while they are allowed for the

TABLE VII

RESULTS OF THE HEURISTICS EVOLVED ON THREE CLASSES; WHEN TESTED ON BENCHMARK INSTANCES OF ALL CLASSES, THIS TABLE SHOWS THE GENERALITY OF THE EVOLVED HEURISTICS

	Best-Fit		Best Evolved Heuristic			Average of 10 Heuristics		
	Version 1	Version 2	N3+N4+N5	N4+N5+N6	N5+N6+N7	N3+N4+N5	N4+N5+N6	N5+N6+N7
N1	45	45	45	40	43	43.3	42.1	44.1
N2	53	53	54	56	56	54.5	55	56.2
N3	52	52	53	52	52	52.7	52.9	52.3
N4	83	86	82	84	82	83.2	83.3	83.6
N5	105	105	106	105	109	106.2	107.4	107.8
N6	103	102	103	102	104	103	103.1	103.5
N7	107	108	104	103	104	105.6	105.1	104.6
N8	84	83	84	83	83	82.8	82.9	82.9
N9	152	152	157	153	156	156.8	155.2	156.2
N10	152	152	153	153	153	155.8	153.2	154.5

TABLE VIII

RESULTS OF OUR BEST EVOLVED HEURISTICS ON BENCHMARK DATA SETS, COMPARED TO RECENT METAHEURISTIC AND CONSTRUCTIVE HEURISTIC APPROACHES, AND THE STATE OF THE ART REACTIVE GRASP APPROACH

Name	Number of Pieces	Optimal Height	Meta-heuristic	BF Heuristic	Reactive GRASP	Best Evolved	
						Result	Time (s)
N1	10	40	40	45	40	40	<0.01
N2	20	50	51	53	51	56	0.01
N3	30	50	52	52	51	52	0.04
N4	40	80	83	83	81	84	0.12
N5	50	100	106	105	102	105	0.25
N6	60	100	103	103	101	102	0.15
N7	70	100	106	107	101	103	0.39
N8	80	80	85	84	81	83	0.61
N9	100	150	155	152	151	153	0.391
N10	200	150	154	152	151	153	1.09
N11	300	150	155	152	151	152	2.28
N12	500	300	312	306	303	307	4.65
c1p1	16	20	20	21	20	22	0.02
c1p2	17	20	21	22	20	22	0.02
c1p3	16	20	20	24	20	24	0.02
c2p1	25	15	16	16	15	18	0.05
c2p2	25	15	16	16	15	26	0.06
c2p3	25	15	16	16	15	17	0.05
c3p1	28	30	32	32	30	32	0.06
c3p2	29	30	32	34	31	34	0.12
c3p3	28	30	32	33	30	36	0.09
c4p1	49	60	64	63	61	63	0.30
c4p2	49	60	63	62	61	62	0.31
c4p3	49	60	62	62	61	63	0.25
c5p1	73	90	94	93	91	92	0.47
c5p2	73	90	95	92	91	93	0.59
c5p3	73	90	95	93	91	93	0.53
c6p1	97	120	127	123	121	123	1.19
c6p2	97	120	126	122	121	122	1.23
c6p3	97	120	126	124	121	123	1.12
c7p1	196	240	255	247	244	244	6.34
c7p2	197	240	251	244	242	244	7.72
c7p3	196	240	254	245	243	245	7.64
NiceP1	25	100	108.2	107.4	103.7	108.9	0.06
NiceP2	50	100	112	108.5	104.6	110.1	0.33
NiceP3	100	100	113	107	104	108.1	1.97
NiceP4	200	100	113.2	105.3	103.6	107.5	10.59
NiceP5	500	100	111.9	103.5	102.2	104.4	110.5
NiceP6	1000	100	–	103.7	102.2	104.1	654.1
PathP1	25	100	106.7	110.1	104.2	111.0	0.08
PathP2	50	100	107	113.8	101.8	106.5	0.45
PathP3	100	100	109	107.3	102.6	104.3	3.34
PathP4	200	100	108.8	104.1	102	104.1	19.44
PathP5	500	100	111.11	103.7	103.1	103.5	194.70
PathP6	1000	100	–	102.8	102.5	104.9	1207.85
RBP1	50	375	400	400	375	400	0.06

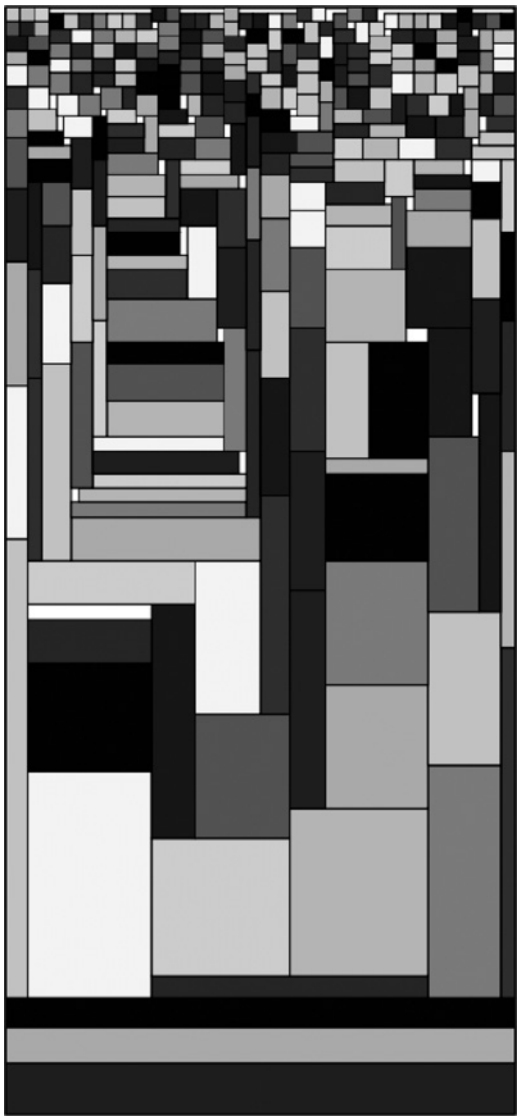

Fig. 14. Packing obtained by the best evolved heuristic on instance N11, to a height of 152. In contrast to its usual packing strategy, the first three pieces are laid horizontally, perhaps because they fit exactly into the width of the slot.

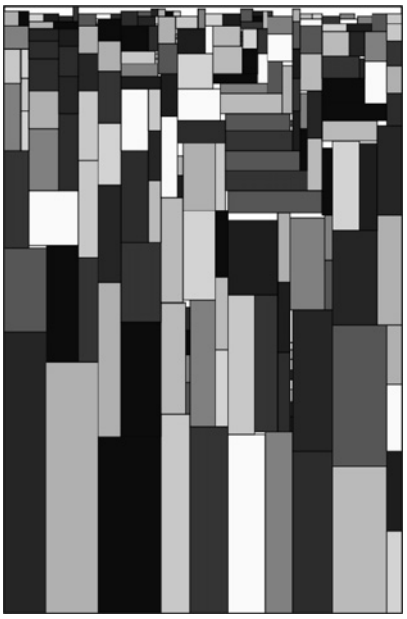

Fig. 15. Packing obtained by the best evolved heuristic on instance c7p2, to a height of 244. Note that if we applied a postprocessing stage such as the one used by best-fit, the small piece at the top would be laid flat and the solution would be one unit better.

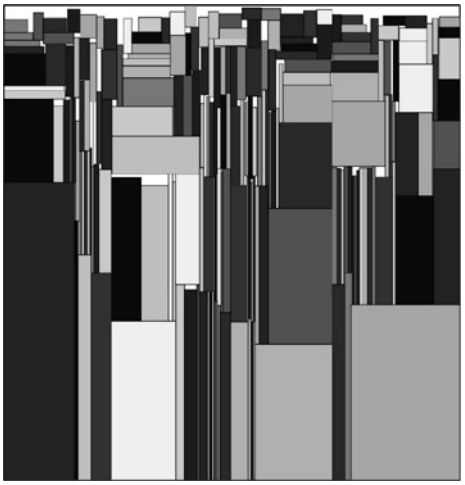

Fig. 16. Packing obtained by the best evolved heuristic on instance PathP4, to a height of 104.1. Similar to Fig. 15, postprocessing would improve this solution further, by laying the tallest piece on its side.

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.

The table shows that the automatically designed heuristic has a performance roughly the same as the best-fit heuristic. It is noticably better than the metaheuristic methods, and noticably worse than the reactive GRASP approach. This is an appropriate place for the evolved heuristic, as the reactive GRASP method is the state of the art in 2-D strip packing, and is a complex algorithm with many parameters, which are chosen by hand to enable the algorithm to obtain the best results in the literature. We would not expect simple constructive heuristics to perform better than such a method, whether they are designed by hand or by GP.

The "time" column displays the time that the heuristic takes to produce a solution. The run times are worth noting for the two largest instances from each of the nice and path sets. This is due to the methodology of iterating through all of the possible piece and slot combinations at every decision point. These instances have two characteristics that result in very large run times. The first is that the sheet width is large compared to the average width of the pieces. This means that many more slots are created, as more pieces fit into the sheet width, and each potentially creates a new slot. This is combined with the fact that the heuristic prefers to place long thin pieces vertically, which creates more slots as all of the pieces stack up next to each other, and when their heights do not quite match up, each one will produce a separate slot. In this situation, the heuristic is evaluated a significant number of times more than is necessary. If the exact strategy of the

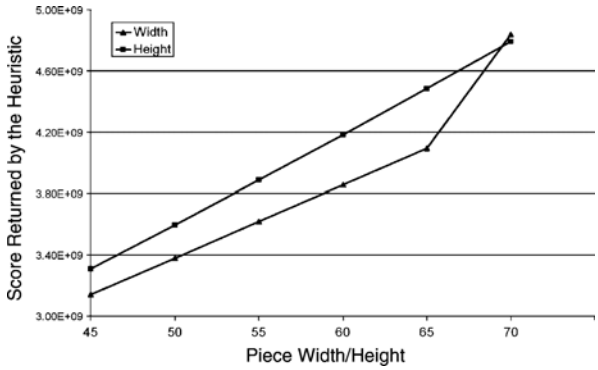

Fig. 17. Scores returned by the heuristic for different piece sizes in the first slot of N11. For the "width" plot, the width is increased to 70 as the height is fixed to 20. The converse is true for the "height" plot.

evolved heuristic can be extracted and reimplemented, then the process of packing can be made much more efficient, and this forms part of our future work (see Section VIII for more discussion of this issue).

1) *Example Packings:* Three example packings, obtained by the best evolved heuristic, are shown in Figs. 14–16. They show the scalability of the heuristic, as it was trained on instances with 40–60 pieces, and the instances shown contain 197–300 pieces.

The heuristic has a tendency to pack pieces vertically rather than horizontally, especially at the beginning of the packing, and this behavior can be seen in Figs. 15 and 16. Indeed, it is this behavior that results in the very poor solution to instance c2p2 (see Table VIII), which is very wide compared to its optimal height, and which contains one very long piece which must be laid horizontally. In contrast, Fig. 14 shows the first three pieces packed horizontally, and the reason for this can be seen in the graph in Fig. 17.

In Fig. 17, the two lines represent the scores returned by the heuristic for the first slot in N11, for different sized pieces. The first line represents the score returned by the heuristic when the piece height increases to 70 (the width is fixed to 20). The second line represents the score returned by the heuristic when the width increases to 70 (the height is fixed to 20). These two lines represent possibilities for placing pieces into the first slot, and the scores that the heuristic gives to those possibilities. The two lines represent the fact that pieces can take two orientations, and we only extend the dimensions up to 70 because the width of the sheet is 70.

One can see from the width line that the score increases in a linear fashion until the width hits 70, in which case the score increases dramatically, taking it above the score when the height is 70. A possible reason for this is that when the piece fits a slot exactly, the "SWL" terminal takes a value of zero, and this may render sections of the heuristic redundant, especially if they involve a multiplication with SWL.

This means that the heuristic scores the piece higher in its horizontal orientation than its vertical orientation. In the absence of any other pieces in this instance with a dimension greater than 70, this piece receives the highest score in its horizontal orientation. If there existed pieces with a height greater than 70, these would receive an even higher score, because the line in Fig. 17 representing height would extend to its next point at a height of 75. Thus, the reason for the heuristic placing pieces vertically at the beginning of the packing is that there rarely exists a piece that fits the width of the sheet exactly, as is the case in instance N11.

VII. Conclusion

Traditionally, heuristics have been human designed, which is a highly appropriate approach for many situations, especially where the importance of obtaining a solution close to the optimum is paramount. However, there are situations where the cost of employing a human heuristic designer may be too high, and where a solution which is as close to the optimum as possible is not required. It is in these situations that the solutions are often obtained by hand, because the cost of a computer aided decision support system is too high. In these situations, it is less important that the solution quality is as close to optimal as possible, and more important that the solutions are simply better than that currently obtained without computer support. Organizations with such a goal would benefit from this type of methodology, where the cost of a heuristic for their problem would be made cheaper through automation of the heuristic design process.

This paper has shown that an evolutionary hyper-heuristic approach can automatically generate very good quality reusable heuristics for the 2-D strip packing problem. The approach represents a change in the way that evolutionary approaches are employed for this problem, and represents the first attempt at automated heuristic design for this problem. The contribution of this paper is not to show that this methodology obtains better results than the best in the literature, or that it can obtain results more quickly, although the results of the evolved heuristics are highly competitive with the best human created constructive heuristic in the literature. The contribution is to show that the design process can be automated for this problem with evolutionary computation, and show the quality of the heuristics that can be designed by evolution.

VIII. Future Work

In practical real world situations where variants of the 2-D strip packing problem occur, the problem instances will not be constructed from a known optimum in which the pieces fit neatly together. The instances will often contain a few types of pieces, with lots of copies of each piece. This is because one organization will produce the same product many times, which will require many copies of identical pieces of material. We hypothesize that this methodology will excel in such a situation. We have already shown that the heuristics can be specialized to a class of problems where the pieces are not identical, and so we believe this phenomenon will be more pronounced if the instances are even more specialized. We intend to test this by creating instances with few piece types, but many copies of each, and testing the quality of the solutions produced.

Another potential research direction would be to determine whether the existing functions and terminals represent the best set for evolving generalizable heuristics, or if they need to be modified to incorporate more general information. For example, the "piece width" terminal currently encodes the absolute value of the width of the piece, but it may be necessary to redefine this terminal. The redefined terminal may encode the piece width as a fraction of the sheet width, or as a fraction of the maximum piece width in the instance. Then the heuristic may be more applicable to new problem instances, and be easier to interpret and understand. To express the issue a different way, if one was to take a problem instance, and create a new instance by reducing the size of all the dimensions by half, then one would expect that applying a heuristic to both instances would produce two solutions that look identical. Currently, because the terminals encode absolute values, it is not clear whether an evolved heuristic would produce identical results for instances which are scaled up or down.

This GP methodology is costly to produce an immediate solution, and while the aim is not to evolve solutions to individual instances, we would still wish to make the process as efficient as possible. The reason for the lengthly run times is a combination of code bloat and the many times that the GP tree must be evaluated for each packing. We use the tarpeian wrapper method [63] to reduce bloat, but this is a general solution that may not be the most effective for this problem. We aim to investigate if there are methods more specific to 2-D packing which can reduce the redundant code, without compromising the variety of the heuristics in the population.

Once they are evolved, the heuristics are not optimized, and therefore look slower than existing heuristics such as best-fit. We would like to investigate the possibility of extracting a method from the evolved tree and then optimize the implementation of it. For example, if it can be shown that the evolved tree always scores lower slots much higher, appropriate data structures can be used to ensure that the lowest slot is obtained in the most efficient way. To give another example, if the pieces with a greater height are always scored more highly then the evolved strategy can be reimplemented as a hand programmed heuristic which preorders the pieces. These reimplementations would remove unnecessary calculations that are sure to make no difference to the result. However, this can only be done once the heuristic strategy has been evolved. The very general process of iterating through every piece and slot looks inefficient when the strategy of an evolved heuristic is examined, but keeping the process very general is necessary, to ensure that it is possible to evolve a variety of strategies.

REFERENCES

[1] P. Ross, "Hyper-heuristics," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Boston, MA: Kluwer, 2005, pp. 529–556.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[3] E. K. Burke, R. S. R. Hellier, G. Kendall, and G. Whitwell, "A new bottom-left-fill heuristic algorithm for the 2-D irregular packing problem," *Oper. Res.*, vol. 54, no. 3, pp. 587–601, 2006.

[4] G. Wäscher, H. Haußner, and H. Schumann, "An improved typology of cutting and packing problems," *Eur. J. Oper. Res.*, vol. 183, no. 3, pp. 1109–1130, 2007.

[5] E. Hopper and B. C. H. Turton, "An empirical investigation of meta-heuristic and heuristic algorithms for a 2-D packing problem," *Eur. J. Oper. Res.*, vol. 128, no. 1, pp. 34–57, 2001.

[6] S. Jakobs, "On genetic algorithms for the packing of polygons," *Eur. J. Oper. Res.*, vol. 88, no. 1, pp. 65–181, 1996.

[7] D. Whitley and J. P. Watson, "Complexity theory and the no free lunch theorem," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Boston, MA: Kluwer, 2005, pp. 317–339.

[8] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[9] D. Whitley and J. P. Watson, "Complexity and no free lunch," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Boston, MA: Kluwer, 2005, pp. 317–339.

[10] E. K. Burke and S. Petrovic, "Recent research trends in automated timetabling," *Eur. J. Oper. Res.*, vol. 140, no. 2, pp. 266–280, 2002.

[11] R. Lewis, "A survey of metaheuristic-based techniques for University Timetabling problems," *OR Spectrum*, vol. 30, no. 1, pp. 167–190, 2008.

[12] Z. P. Lu and J. K. Hao, "Adaptive tabu search for course timetabling," *Eur. J. Oper. Res.*, vol. 200, no. 1, pp. 235–244, 2010.

[13] R. Qu, E. K. Burke, B. McCollum, L. G. T. Merlot, and S. Y. Lee, "A survey of search methodologies and automated system development for examination timetabling," *J. Scheduling*, vol. 12, no. 1, pp. 55–89, 2009.

[14] E. K. Burke, D. G. Elliman, P. H. Ford, and R. F. Weare, "Examination timetabling in British universities—A survey," in *The Practice and Theory of Automated Timetabling* (Lecture Notes in Computer Science 1153), E. K. Burke and P. Ross, Eds. Berlin, Germany: Springer, 1996, pp. 76–92.

[15] K. Schimmelpfeng and S. Helber, "Application of a real-world university-course timetabling model solved by integer programming," *OR Spectrum*, vol. 29, no. 4, pp. 783–803, 2007.

[16] A. S. Fukunaga, "Automated discovery of composite SAT variable-selection heuristics," in *Proc. 18th Natl. Conf. Artif. Intell.*, Menlo Park, CA: American Association for Artificial Intelligence, 2002, pp. 641–648.

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

[18] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Boston, MA: MIT Press, 1992.

[19] J. R. Koza and R. Poli, "Genetic programming," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Boston, MA: Kluwer, 2005, pp. 127–164.

[20] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming, an Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA: Morgan Kaufmann, 1998.

[21] P. Ross, J. G. Marin-Blazquez, S. Schulenburg, and E. Hart, "Learning a procedure that can solve hard bin-packing problems: A new GA-based approach to hyperheuristics," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, Chicago, IL, 2003, pp. 1295–1306.

[22] P. Ross, S. Schulenburg, J. G. Marin-Blazquez, and E. Hart, "Hyper heuristics: Learning to combine simple heuristics in bin packing problems," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, New York, 2002, pp. 942–948.

[23] L. Han and G. Kendall, "Investigation of a tabu assisted hyper-heuristic genetic algorithm," in *Proc. Congr. Evol. Comput. (CEC)*, vol. 3. Canberra, Australia, Dec. 2003, pp. 2230–2237.

[24] E. K. Burke, S. Petrovic, and R. Qu, "Case-based heuristic selection for timetabling problems," *J. Schedul.*, vol. 9, no. 2, pp. 115–132, 2006.

[25] K. Dowsland, E. Soubeiga, and E. K. Burke, "A simulated annealing hyper-heuristic for determining shipper sizes," *Eur. J. Oper. Res.*, vol. 179, no. 3, pp. 759–774, 2007.

[26] E. K. Burke, G. Kendall, and E. Soubeiga, "A tabu-search hyper-heuristic for timetabling and rostering," *J. Heurist.*, vol. 9, no. 6, pp. 451–470, 2003.

[27] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper heuristic for timetabling problems," *Eur. J. Oper. Res.*, vol. 176, no. 1, pp. 177–192, Jan. 2007.

[28] E. Özcan, B. Bilgin, and E. E. Korkmaz, "Hill climbers and mutational heuristics in hyperheuristics," in *Proc. 9th Int. Conf. Parallel Problem Solving Nature (PPSN)*, LNCS 4193. Reykjavik, Iceland, Sep. 2006, pp. 202–211.
[29] P. Cowling, G. Kendall, and E. Soubeiga, "A hyperheuristic approach to scheduling a sales summit," in *Proc. 3rd Int. Conf. Practice Theory Automated Timetabling (PATAT)*, Konstanz, Germany, Aug. 2000, pp. 176–190.
[30] P. Rattadilok, A. Gaw, and R. Kwan, "Distributed choice function hyper-heuristics for timetabling and scheduling," in *Proc. 5th Int. Conf. Practice Theory Automated Timetabling (PATAT)*, LNCS 3616. 2005, pp. 51–67.
[31] A. S. Fukunaga, "Evolving local search heuristics for SAT using genetic programming," in *Proc. Assoc. Comput. Machinery Genetic Evol. Comput. Conf. (GECCO)*, LNCS 3103. Seattle, WA: Springer-Verlag, 2004, pp. 483–494.
[32] A. Fukunaga, "Automated discovery of local search heuristics for satisfiability testing," in *Evol. Comput.*, vol. 16, no. 1, pp. 31–61, 2008.
[33] M. B. Bader-El-Din and R. Poli, "Generating SAT local-search heuristics using a GP hyper-heuristic framework," in *Proc. 8th Int. Conf. Artif. Evol.*, LNCS 4926. Oct. 2007, pp. 37–49.
[34] E. K. Burke, M. R. Hyde, and G. Kendall, "Evolving bin packing heuristics with genetic programming," in *Proc. 9th Int. Conf. Parallel Problem Solving Nature (PPSN)*, Reykjavik, Iceland, Sep. 2006, pp. 860–869.
[35] 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 *Proc. 9th Assoc. Comput. Machinery Genetic Evol. Comput. Conf. (GECCO)*, London, U.K., Jul. 2007, pp. 1559–1565.
[36] E. K. Burke, M. Hyde, G. Kendall, and J. Woodward, "The scalability of evolved on line bin packing heuristics," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Singapore, Sep. 2007, pp. 2530–2537.
[37] R. E. Keller and R. Poli, "Linear genetic programming of parsimonious metaheuristics," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Singapore, Sep. 2007, pp. 4508–4515.
[38] C. D. Geiger, R. Uzsoy, and H. Aytug, "Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach," *J. Schedul.*, vol. 9, no. 1, pp. 7–34, 2006.
[39] P. Gilmore and R. Gomory, "Multistage cutting stock problems of two and more dimensions," *Oper. Res.*, vol. 13, no. 1, pp. 94–120, 1965.
[40] N. Christofides and C. Whitlock, "An algorithm for 2-D cutting problems," *Oper. Res.*, vol. 25, no. 1, pp. 30–44, 1977.
[41] J. E. Beasley, "An exact 2-D nonguillotine cutting tree search procedure," *Oper. Res.*, vol. 33, no. 1, pp. 49–64, 1985.
[42] M. Hifi and V. Zissimopoulos, "A recursive exact algorithm for weighted 2-D cutting," *Eur. J. Oper. Res.*, vol. 91, no. 3, pp. 553–564, 1996.
[43] V. D. Cung, M. Hifi, and B. Le Cun, "Constrained 2-D cutting stock problems: A best-first branch-and-bound algorithm," *Int. Trans. Oper. Res.*, vol. 7, no. 3, pp. 185–210, 2000.
[44] F. Clautiaux, A. Jouglet, J. Carlier, and A. Moukrim, "A new constraint programming approach for the orthogonal packing problem," *Comput. Oper. Res.*, vol. 35, no. 3, pp. 944–959, 2008.
[45] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, and H. Nagamochi, "Exact algorithms for the 2-D strip packing problem with and without rotations," *Eur. J. Oper. Res.*, vol. 198, no. 1, pp. 73–83, 2009.
[46] R. Macedo, C. Alves, and J. M. V. de Carvalho, "Arc-flow model for the 2-D guillotine cutting stock problem," *Comput. Oper. Res.*, vol. 37, no. 6, pp. 991–1001, 2010.
[47] R. Alvarez-Valdes, F. Parreno, and J. M. Tamarit, "A branch and bound algorithm for the strip packing problem," *OR Spectrum*, vol. 31, no. 2, pp. 431–459, 2009.
[48] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest, "Orthogonal packings in two dimensions." *Soc. Ind. Appl. Math. J. Comput.*, vol. 9, no. 4, pp. 846–855, 1980.
[49] B. Chazelle, "The bottom-left bin packing heuristic: An efficient implementation," *IEEE Trans. Comput.*, vol. 32, no. 8, pp. 697–707, Aug. 1983.
[50] D. J. Brown, B. S. Baker, and H. P. Katseff, "Lower bounds for on-line 2-D packing algorithms," *Acta Inform.*, vol. 18, no. 2, pp. 207–226, 1982.
[51] D. Zhang, Y. Kang, and A. Deng, "A new heuristic recursive algorithm for the strip rectangular packing problem," *Comput. Oper. Res.*, vol. 33, no. 8, pp. 2209–2217, 2006.
[52] F. Rinaldi and A. Franz, "A 2-D strip cutting problem with sequencing constraint," *Eur. J. Oper. Res.*, vol. 183, no. 3, pp. 1371–1384, 2007.
[53] E. K. Burke, G. Kendall, and G. Whitwell, "A new placement heuristic for the orthogonal stock-cutting problem," *Oper. Res.*, vol. 55, no. 4, pp. 655–671, 2004.
[54] E. K. Burke, G. Kendall, and G. Whitwell, "A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock cutting problem," *Inst. Oper. Res. Manage. Sci. J. Comput.*, vol. 21, no. 3, pp. 505–516, 2009.
[55] D. Liu and H. Teng, "An improved BL-algorithm for genetic algorithms of the orthogonal packing of rectangles," *Eur. J. Oper. Res.*, vol. 112, no. 2, pp. 413–419, 1999.
[56] G. Belov, G. Scheithauer, and E. A. Mukhacheva, "1-D heuristics adapted for 2-D rectangular strip packing," *J. Oper. Res. Soc.*, vol. 59, no. 6, pp. 823–832, 2008.
[57] A. Ramesh Babu and N. Ramesh Babu, "Effective nesting of rectangular parts in multiple rectangular sheets using genetic and heuristic algorithms," *Int. J. Prod. Res.*, vol. 37, no. 7, pp. 1625–1643, 1999.
[58] C. L. Valenzuela and P. Y. Wang, "VLSI placement and area optimization using a genetic algorithm to breed normalized postfix expressions," *IEEE Trans. Evol. Comput.*, vol. 6, no. 4, pp. 390–401, Aug. 2002.
[59] K. K. Lai and J. W. M. Chan, "Developing a simulated annealing algorithm for the cutting stock problem," *Comput. Ind. Eng.*, vol. 32, no. 1, pp. 115–127, 1997.
[60] L. Faina, "An application of simulated annealing to the cutting stock problem," *Eur. J. Oper. Res.*, vol. 114, no. 3, pp. 542–556, 1999.
[61] A. Bortfeldt, "A genetic algorithm for the 2-D strip packing problem with rectangular pieces," *Eur. J. Oper. Res.*, vol. 172, no. 3, pp. 814–837, 2006.
[62] R. Alvarez-Valdes, F. Parreno, and J. M. Tamarit, "Reactive grasp for the strip-packing problem," *Comput. Oper. Res.*, vol. 35, no. 4, pp. 1065–1083, 2008.
[63] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in *Proc. 6th Eur. Conf. Genetic Program. (EuroGP)*, C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds. Essex, U.K.: Springer-Verlag, Apr. 2003, pp. 211–223.

Edmund K. Burke (M'06) received the B.Ed. (Hons.) degree in mathematics from the University of Leeds, Leeds, U.K., in 1986, the M.Sc. degree in pure mathematics from the University of Leeds, in 1987, and the Ph.D. degree in computer science from the same university, in 1991.

He is currently the Dean of the Faculty of Science, University of Nottingham, Nottingham, U.K., where he also leads the Automated Scheduling, Optimization and Planning Research Group, in the School of Computer Science. He has played a leading role in the organization of several major international conferences in his research field in the last few years. He has edited and authored 14 books and has published over 200 refereed papers. His current research interests include the interface of operational research and computer science, in particular, exploring search methodologies and the role that they play in underpinning intelligent decision support systems across a wide variety of applications.

Dr. Burke is a Member of the Engineering and Physical Sciences Research Council (EPSRC) Strategic Advisory Team for Mathematics. He is a Fellow of the Operational Research Society and the British Computer Society, and he is a Member of the U.K. Computing Research Committee. He is the Editor-in-Chief of the *Journal of Scheduling*, the Area Editor (for Combinatorial Optimization) of the *Journal of Heuristics*, an Associate Editor of the *INFORMS Journal on Computing*, an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, and a Member of the Editorial Board of *Memetic Computing*. He has been awarded 54 externally funded grants worth over £13M from a variety of sources, including the EPSRC, the Economic and Social Research Council, the Biotechnology and Biological Sciences Research Council, the European Union, the Research Council of Norway, the East Midlands Development Agency, the Higher Education Funding Council for England, the Teaching Company Directorate, the Joint Information Systems Committee of the Higher Education Funding Councils, and commercial organizations. This funding portfolio includes being the Principal Investigator on an EPSRC Science and Innovation Award of £2M, an EPSRC Grant of £2.6M to investigate the automation of the heuristic design process, and an EPSRC Platform Renewal Grant worth over £1M.

Matthew Hyde (M'07) received the B.S.(Hons.) (First Class) degree in computer science from the University of Nottingham, Nottingham, U.K., in 2005, and a Ph.D. degree in computer science from the University of Nottingham, in 2009.

He is currently a Research Fellow with the Automated Scheduling, Optimization, and Planning Research Group, in the School of Computer Science at the University of Nottingham. He has worked on two Engineering and Physical Sciences Research Council funded projects, of which the first was to investigate genetic programming as a hyper-heuristic. He is currently working with a £2.6M project which aims to investigate methodologies suitable to automate the heuristic design process. He has served on the Program Committee for the 2007 Conference on Evolutionary Computation, and has reviewed papers for four international journals. He has published four papers and one book chapter on the subject of automatic heuristic generation. His research interests include evolutionary computation, hyper-heuristics, metaheuristics, and operational research.

Graham Kendall (M'03) received the B.S.(Hons.) (First Class) degree in computation from the University of Manchester Institute of Science and Technology (UMIST), Manchester, U.K., in 1997, and the Ph.D. degree from the School of Computer Science, University of Nottingham, Nottingham, U.K., in 2000.

Before entering academia he spent almost 20 years in the IT industry, working for various U.K. companies, taking on roles such as Computer Operator, Technical Support Manager, and Service Manager. He is currently the Dunford Professor of computer science with the Automated Scheduling, Optimization and Planning Research Group, School of Computer Science, University of Nottingham. He has edited and authored nine books and has published over 100 refereed journal and conference papers. His current research interests include scheduling, optimization, evolutionary and adaptive learning, heuristics and meta/hyper heuristics. He has a particular interest in solving real world problems.

Prof. Kendall is a Fellow of the Operational Research Society and an Associate Editor of seven international journals. He chairs the Steering Committee of the Multidisciplinary International Conference on Scheduling: Theory and Applications, in addition to having chaired several other international conferences in recent years. He has been a Member of the program/technical committees of over 130 international conferences over the last ten years. He has been awarded externally funded grants worth over £5M from a variety of sources, including the Engineering and Physical Sciences Research Council and commercial organizations.

John Woodward received the B.S. degree in theoretical physics, in 1989, the M.S. degree in cognitive science (with distinction), in 1997, and the Ph.D. degree in computer science from the University of Birmingham, in the U.K., in 2005.

He recently completed a Post-Doctoral position at the University of Nottingham, where he investigated the use of genetic programming to discover novel heuristics. In addition, he was with the European Organization for Nuclear Research, where he did research into particle physics, the Royal Air Force as an Environmental Noise Scientist, and Electronic Data Systems as a Systems Engineer. Currently, he is with the School of Computer Science, at the University of Nottingham, Ningbo Campus, in China, where he is a Member of the Automated Scheduling, Optimization and Planning Research Group, His research interests include fundamental issues in machine learning, especially genetic programming.