A Histogram-matching Approach to the Evolution of Bin-packing Strategies

Riccardo Poli John Woodward Edmund K. Burke

Abstract— **We present a novel algorithm for the one-dimension offline bin packing problem with discrete item sizes based on the notion of matching the item-size histogram with the bin-gap histogram. The approach is controlled by a constructive heuristic function which decides how to prioritise items in order to minimise the difference between histograms. We evolve such a function using a form of linear register-based genetic programming system. We test our evolved heuristics and compare them with hand-designed ones, including the well-known best fit decreasing heuristic. The evolved heuristics are human-competitive, generally being able to outperform high-performance human-designed heuristics.**

I. Introduction

We consider the one-dimensional off-line bin packing problem. The aim is to pack a set of items, of various sizes, into identical bins of a given fixed capacity, such that no bin's contents overflows. The objective is to minimise the number of bins used. In essence, we are partitioning the set of items into a minimum number of subsets such that the total size of items in each subset does not exceed the capacity of a bin. This is one of the most studied problems in the computer science literature and has vast industrial applications [6].

The problem of finding an optimal packing is known to be NP-hard [3], so heuristics are used to find near optimal solutions. In fact, *"almost all the papers published on the bin packing problem concern heuristic algorithms"* [9]. A good human-designed heuristic is First Fit Decreasing (FFD), which first sorts items by non-increasing size and then places them, one at a time, into the first bin in which they fit. A variant of this, known as Best Fit Decreasing (BFD), places items into the bin where they fit most tightly. These are two of the fastest heuristics (see [9] for an overview). Unfortunately, these algorithms perform poorly when the solutions demand that most of the bins are nearly full. In [6] a heuristic algorithm is developed which is optimal, in the rather special cases, where the bin capacity is at least half the total capacity required to pack all of the items or, the optimal solution requires that most of the bins are nearly full. It is also possible to create instances for which this minimum bin slack heuristic [6] will spend hours before finding a solution [5]. [5] introduce four heuristics based on the minimum bin slack heuristic. It is all very well if the heuristic is suited to class of problem at hand, but the difficulty lies in designing heuristics for classes of problem which heuristics are difficult to design by hand. The objective of this paper is to automatically achieve this using Genetic Programming (GP) [7], [8]. Some heuristics can be "tricked", as removing an item from the set of items should obviously not increase the number of bins used (as there are fewer items), however this is not the case with some heuristics [2].

An alternative to designing heuristics by hand would be to apply a meta heuristic (for example, a genetic algorithm) to tackle a given instance of the bin packing problem. In this case, the genotype would consist of information which, when interpreted, assigns a given item to a given bin *explicitly* (e.g., item 4 goes into bin 29). Falkenauer [4] discusses some of the issues surrounding representation and introduces a hybrid grouping genetic algorithm. However, *"Although the evolutionary algorithm may solve a given problem very well, simply re-running the EA or changing the problem slightly may produce very different and/or worse results"* [11]. In addition, further parameter tuning may be necessary. For this reason we consider hyper heuristics (i.e., heuristics to choose heuristics) [2]. In [11], a hyper heuristic approach is used. A number of heuristics are used in order to combine them into a system which performs better than any of the constituent heuristics. Impressive results are obtained, but the question arises as to how these heuristics are created in the first place.

Human-designed heuristics perform well under certain circumstances for which they are specifically tailored, but perform poorly under other circumstances dictated by the distribution of the item sizes. In general, the relationship between the problem class, which is defined by the distribution of item sizes, and which heuristic will perform better, is complex and often unknown. The automated design of effective heuristics via evolution is, therefore, an appealing perspective. For example, the use of GP would allow the discovery of heuristics which are difficult for humans to find as the space of human designs is a subset of the space of all designs [10, page 2]. This motivates us to build systems which can automatically generate heuristic algorithms for the specific, but possibly ill-defined, class of problems at hand.

In [5] four variants of the minimum bin slack heuristic and a framework for variable neighbourhood search are presented. The hope is that due to this flexible framework, it may be possible to design even more sophisticated heuristic algorithms. Our suggestion is to use this as a framework for a search space in which GP can evolve designs.

There are a number of motivations for evolving heuristics. Firstly, we can construct a space which contains all of the heuristic algorithms including the human design heuristics mentioned above. The system can then search for an algorithm, rather than having humans generate them. We are therefore automating the design process of heuristic algorithms. If the representation used to construct the search space is capable of expressing human designed heuristics, then it is possible that the system should at least be able to

Riccardo Poli is with the Department of Computer Science, University of Essex, UK (email: rpoli@essex.ac.uk). John Woodward and Edmund Burke are with the department of Computer Science and IT, University of Nottingham (email: jrw, ekb@cs.nott.ac.uk).

achieve human competitiveness. Secondly, once a heuristic algorithm is obtained, it can be applied to new problems without having to go through the process of searching the space again, which is unlike the situation with the meta heuristic approach (for example, if we were using a GA approach, where items are hard-coded into the genotype, this does not help with future problems). We are attempting to find *general* solutions to a class of problem, rather than a *specific* solution to a single problem. Finally, some human designed heuristics perform well on problem classes which are simple to describe [6]. However, given a class of problem which has a distribution of item sizes which is difficult to describe, it is hoped that GP will be able to produce a heuristic algorithm which performs well on this type of distribution. Although the item size distribution in the bin packing problem is typically a uniform distribution between 0 and the bin capacity, in most real world industrial applications this will not be the case. In [6] they mention tackling more complex bin packing environments, and using GP is one way to do this empirically, which is where the interests of most industrial applications lie.

In previous work [1], a tree-based GP system for *online* bin-packing was presented. This evolves a heuristic that decides whether to put an item in a bin when presented with the sum of the sizes of items already in the bin, and the size of the item that is about to be packed. This heuristic operates in a fixed framework that iterates through the open bins, applying the heuristic to each one, before deciding which bin to use. The best evolved programs emulate the functionality of the human designed first-fit heuristic. In this paper, thanks to a novel approach to bin packing used in conjunction with a linear GP system, we are able to evolve human-competitive offline heuristics bin packing with discrete item sizes. On non-uniform distributions of item sizes, these outperform to a significant extent the well-known BFD heuristic.

The paper is organised as follows. In Section II we describe a novel approach to bin packing based on the notion of matching the item size histogram with the bin-gap histogram. In Section III we describe four hand-designed heuristics and show their relation to BFD. In Section IV we describe our GP system, including its fitness function, operators and training set. In Section V we test and compare the performance of the hand-designed and evolved heuristics. Finally, we draw some conclusions in Section VI.

II. Bin packing via histogram matching

Let S be the bin size. Instead of focusing on how full the bins are, we imagine that the space remaining (i.e., the gap) can vary dynamically. For example, if a bin contains items totalling size $\hat{s}$, we will just think of it as a gap of size $S - \hat{s}$. Further, let us restrict our attention to the case where both S and item sizes are integers. So, gap sizes are also integers.

Let $g_s(t)$ represent the number of gaps of size s at time t, where time is measured in terms of the number of items packed. Naturally, $g_s(0) = 0$ for all s. Let $o_s(t)$ be the number of items of size s which still need to be packed at time t. Thus, $g_s(t) = 0$ and $o_s(t)$ can be thought of as histograms. Let $O = \sum_s o_s(0)$ be the total number of items that need to be packed. Clearly, $\sigma = \sum_s s \times o_s(0)$ is the minimal volume required to pack all the items. So, the minimum number of bins required to solve the problem is $B_{min} = \lceil \sigma / S \rceil$, while the maximum is O, corresponding to one item in each bin.

The objective of the histogram matching algorithm is to get $g_s(t) \geq o_s(t)$ (where comparison is extended component by component) at some time t, for all s, with equality holding for as many s as possible. If this state can be achieved, the problem is solved, the algorithm can stop and we know that there is a gap of appropriate size in the bins for each of the remaining items awaiting to be packed.

Let us consider an example. Suppose we need to pack two items of size 2, one of size 3 and two of size 4, i.e., $o_s(0) = (0, 2, 1, 2)$, and that the bin size is $S = 6$. Naturally, $g_s(0) = (0, 0, 0, 0)$ since there are no bins currently open. Let us assume we want to start by packing an item of size 4. Obviously, we need to place the item in a new bin. So, the current bin configuration is $\lfloor\, 4 \,\rfloor$. This move changes the item histogram as follows: $o_s(1) = (0, 2, 1, 1)$. Of course, the insertion of an item of size 4 in a bin of size 6 produces a gap of size 2. Therefore, we also have that $g_s(1) = (0, 1, 0, 0)$. Suppose we then decide to pack another item of size 4. Since the gap in the first bin is too small to hold it, we will need to start a further bin, giving: $o_s(2) = (0, 2, 1, 0)$ and $g_s(2) = (0, 2, 0, 0)$. So, the current bin configuration is $\lfloor\, 4 \,\|\, 4 \,\rfloor$. Note that there is already a partial match between g_s and o_s, in that $g_1(2) \geq o_1(2)$, $g_2(2) \geq o_2(2)$ and $g_4(2) \geq o_4(2)$. Now suppose we place the item of size 3. Again, this requires a new bin and produces a gap of size 3, leading to $o_s(3) = (0, 2, 0, 0)$ and $g_s(3) = (0, 2, 1, 0)$. So, the current bin configuration is $\lfloor\, 4 \,\|\, 4 \,\|\, 3 \,\rfloor$. Now, $g_s \geq o_s$ for all s. This means that there are sufficient gaps to place all remaining items. Placing such items produces the final gap histogram $g_s(3) - o_s(3) = (0, 0, 1, 0)$. So, we have used three bins and the final bin configuration is $\left\lfloor \begin{matrix}2\\4\end{matrix} \,\middle\|\, \begin{matrix}2\\4\end{matrix} \,\middle\|\, 3 \right\rfloor$.

There are many strategies by which one can achieve the target state $g_s(t) \geq o_s(t)$. A simple algorithm to do this is shown in Algorithm 1. This algorithm receives S and $o_s(0)$ as input and achieves the target state by iteratively changing $o_s(t)$ and $g_s(t)$. The algorithm works as follows. It iterates its main body (lines 2–21) until the list of sizes for which $g_s < o_s$ (a condition that we term a "clash") is empty. If this is not the case, then the algorithm chooses a new item to pack (line 6). This choice is performed by the function **selectitem**() on the basis of the list of clashes as well as the item and gap histograms. The algorithm then identifies, out of all the possible gaps currently available, those which could be filled with the selected item (lines 7 and 8). Not all gaps are considered to be available: only those which are in excess of the need (i.e., $g_s > o_s$). If one or more such gaps exist, then the function **selectgap**() is invoked to choose which gap to fill (lines 9 and 10). Once a gap is selected, the gap is filled with the item. This implies updating the gap histogram (lines 11–14). If no appropriate gaps exist, then a new bin is opened (line 16), and the gap histogram is updated (lines 17–19). Finally, the item histogram (line 21) is updated to reflect the packing of the selected item.

III. Hand-designed heuristics

In this first exploration we decided to keep **selectgap**() fixed and, instead, vary the function **selectitem**(). In particular, we choose

$$\mathbf{selectgap}() = \texttt{first(avail-gaps-big-enough)} \quad (1)$$

Effectively, whenever there are multiple gaps that could host an item, this strategy uses the smallest one, thereby preserving the bigger gaps for big items yet to be packed. These are intuitively more difficult to place, and this strategy helps in packing them.

The efficient packing of large items is often critical in determining the overall performance of bin-packing algorithms. So, one might wonder if giving priority to their handling could produce good results. A simple way to achieve this is to use the following definition for the **selectitem** function:

$$\mathbf{selectitem}() = \texttt{last(clashes)} \quad (2)$$

We call this strategy **LAST**. LAST's behaviour is illustrated in the first column of Figure 1, where various snapshots of $g_s(t)$ and $o_s(t)$ taken during the packing of 500 items of sizes uniformly distributed in the range $[21, 70]$. As one can see, the matching between the two histograms proceeds in an orderly fashion from right (big items) to left, with the item and gap histograms gradually collapsing down towards zero. Note that, after around 350 steps, the histograms match and so the algorithm is stopped, and packing continues by simply allocating each remaining item to one of its corresponding gaps. In the last row of the figure, we show the distribution of gaps after this phase is completed.

Interestingly, with the LAST strategy, our histogram matching bin-packer (Algorithm 1) is effectively equivalent to the BFD algorithm. The equivalence arises for the following reasons. Firstly, when using Equation (2), our algorithm always picks the largest unallocated item first. So, although we do not explicitly sort items, we still pick them in the same order as BFD. Secondly, with BFD, bins are always ordered by occupancy levels (so they are ordered by ascending gap sizes). So, the bin that ends up being used for an item is always the one with the smallest gap out of those where the item could fit, which is exactly what Equation (1) does. So, LAST and BFD lead to exactly the same packing.

An alternative line of reasoning is the following. If the objective of the histogram matching algorithm is to get g_s and o_s to match as quickly as possible, it would make sense to give priority to those sizes for which there is a large difference between the number of available gaps and the number of items of that size. This can be achieved using

$$\mathbf{selectitem}(...) = \arg\min_s (g_s - o_s) \quad (3)$$

We call this strategy **MIN**. The behaviour of MIN is illustrated in the second column of Figure 1. In this case, the matching between the two histograms proceeds more uniformly, with no size being particularly favoured. Also, unlike with LAST, the histograms are not forced to collapse to zero.

A variant of this is one where we still look at how closely histograms match, as in MIN, but we also give higher importance to larger sizes. A possible implementation of this strategy is the following

$$\mathbf{selectitem}(...) = \arg\min_s [s \times (g_s - o_s)] \quad (4)$$

We call this strategy **SMIN**. As shown in Figure 1 (third column) the behaviour of SMIN is somehow half-way between LAST and MIN, with the bigger items being given higher priority, but the histograms not being forced to collapse to zero.

The final hand-designed strategy we consider is much simpler than MIN and SMIN: simply pick a random clash and try to eliminate it. This strategy, which we call **RAND**, is implemented by the following

$$\mathbf{selectitem}(...) = \texttt{random-element(clashes)} \quad (5)$$

With this strategy the matching of the histograms is performed rather uniformly, with some similarity with the MIN strategy.

The MIN and SMIN strategies presented above are both instances of the following more general form:

$$\mathbf{selectitem}(...) = \arg\min_s [f(s) \times (g_s - o_s)] \quad (6)$$

where $f(s) = 1$ in MIN, while $f(s) = s$ in SMIN. So, one may wonder whether there would be other functional forms for $f(s)$ that could provide meaningful strategies.

Algorithm 1 Bin-packing by histogram matching.

```
1: loop
2:    clashes = list of $s$ values for which $g_s < o_s$
3:    if clashes is empty then
4:       STOP
5:    end if
6:    nextitem = selectitem($g_s$,$o_s$,clashes)
7:    available-gaps= list of $s$ values where $g_s > o_s$
8:    avail-gaps-big-enough= subset of available-gaps for
      which $s >=$ nextitem
9:    if avail-gaps-big-enough is not empty then
10:      gap=selectgap($g_s$,$o_s$,avail-gaps-big-enough);
11:      $g_{\text{gap}} = g_{\text{gap}} - 1$
12:      if place != nextitem then
13:         $g_{(\text{gap}-\text{nextitem})} = g_{(\text{gap}-\text{nextitem})} + 1$
14:      end if
15:   else
16:      bincounter ++;
17:      if $S >$ nextitem then
18:         $g_{(\text{S}-\text{nextitem})} = g_{(\text{S}-\text{nextitem})} + 1$
19:      end if
20:   end if
21:   $o_{\text{nextitem}} = o_{\text{nextitem}} - 1$
22: end loop
```

IV. Evolving histogram-matching strategies

Although one can manually design and test many bin-packing strategies, it is very appealing to be able to automate the process of strategy selection. This, for example, would make it possible to adapt one's bin-packer to changes of the distribution of items. In this section, we consider the

TABLE I

PRIMITIVE SET USED TO EVOLVE HISTOGRAM-MATCHING STRATEGIES

Primitive	Explanation
NOP	No operation is performed
A=A+R1	Add content of register R1 to accumulator A
A=A+R2	Add content of register R2 to accumulator A
A=A+R3	Add content of register R3 to accumulator A
A=A+R4	Add content of register R4 to accumulator A
A=A+R5	Add content of register R5 to accumulator A
A=A+B	Add accumulator B to accumulator A
A=A*R1	Multiply A by R1 and store in accumulator A
A=A*R2	Multiply A by R2 and store in accumulator A
A=A*R3	Multiply A by R3 and store in accumulator A
A=A*R4	Multiply A by R4 and store in accumulator A
A=A*R5	Multiply A by R5 and store in accumulator A
A=A*B	Multiply A by B and store in accumulator A
A=sqrt(A)	Take square root of A and store in accumulator A
A=A^2	Square accumulator A
A=A+1	Increment A by 1
A=A+2	Increment A by 2
A=A+1/2	Increment A by 1/2
A=-A	Negate A
B=A	Store A in accumulator B
A=B	Store B in accumulator A
A=A-B	Subtract B from A and store in A
A=R5-R4	Store the difference between R5 and R4 in A
A=abs(A)	Store magnitude of A in A

TABLE II

INPUT REGISTERS

Register	Content
R1	s/S
R2	s_{max}/S
R3	s_{min}/S
R4	o_s
R5	g_s

(s_{max} =largest s for which $o_s > 0$, s_{min} =smallest s for which $o_s > 0$)

evolution of histogram-matching strategies to be used within Algorithm 1. In particular, we consider strategies of the form presented in Equation (6) and we will investigate a GP methodology to evolve the $f(s)$ function.

For this work, we decided to use a form of linear GP. The instruction set includes the primitives described in Table I. Each primitive is internally represented by an integer. A program is therefore represented as a sequence of integers of some prefixed length ℓ. Naturally, ℓ is only an upper bound to the number of instructions actually executed, because of the presence of a NOP operation in the primitive set.

Before execution, the accumulators, A and B, are both initialised to 0. The registers R1 to R5 are protected from overwriting, and are initialised with the values shown in Table II.

After the execution of a program, the value stored in accumulator A is passed through the wrapper $|A| + 10^{-4}$, which guarantees that the result is strictly positive.[1] The result is returned as the value of $f(s)$ associated to one particular size s. Naturally, in order to apply the minimum operation in Equation (6), the execution process must be repeated for $1 \leq s \leq S$.

In order to evolve general heuristics $f(s)$, we used a training set with four different size distributions, with item sizes in the ranges $[20, 80]$, $[1, 80]$, $[30, 70]$ and $[1, 150]$, and with three different numbers of items, 100, 250 and 1,000, for each distribution giving us a total of 12 conditions. In each condition we constructed three random problem instances. So, our training set includes 48 fitness cases. In each fitness case, the distribution of item sizes was approximately flat within the specified ranges.[2]

For each fitness case, we run our histogram-matching bin-packer with its $f(s)$ heuristic implemented via a GP program. Let B_i be the number of bins used by our algorithm to pack the items in fitness case i, and let B_{min_i} be the theoretical minimum. We measure the fitness of a program as

$$\text{Fitness} = \frac{1}{48} \sum_i \left(\frac{B_i}{B_{min_i}} \right) \tag{7}$$

The objective is to minimise this function.

The GP system initialises the population by randomly drawing primitives from the primitive set. This is done uniformly at random except for the primitive NOP which is chosen with a probability 1/3, to provide a more varied distribution of effective code length. The GP system is steady-state and the population is then manipulated by the following operators:

- Tournament selection with tournament size 2.
- All offspring are created by two forms of crossover: a) a form of crossover where the offspring is created by extracting a random section of the first parent and inserting it at a random point in a copy of the second parent; b) uniform crossover (with 50% exchange probability). The form of crossover used to create a particular offspring is determined by an equiprobable Bernoulli trial.
- All offspring undergo point-mutation. This operation is applied with a 0.1 probability (per locus). When a primitive is changed, its replacement is randomly drawn from the primitive set (with the same strategy used for initialisation, i.e., NOPs are more likely than other primitives).

V. EXPERIMENTAL RESULTS

In our experiments, we used programs consisting of $\ell = 10$ instructions, populations of size 100, run for 30 generations. Although we experimented with other parameter settings, with these settings the GP system was able to produce high quality heuristics in virtually all runs. So, there seemed to be no point in using larger programs, population sizes, or runs.

To illustrate the kind of results that the system produces, we report in Table III, two representative evolved heuristics. They are both strictly increasing monotonic functions of s. So, they give precedence to larger items. However, if o_s and/or g_s are non-uniform, then the basic strategy of giving higher priority to bigger sizes is modified, biasing it towards the sizes for which there are many items still to be packed

[1]The reason for having a rectifying wrapper is to ensure that the minimum operation excludes sizes for which g_s is already bigger than o_s.

[2]More precisely, if we denote with $[s_{min}, s_{max}]$ the range of item sizes, our bin size distributions are multinomials, with success probabilities $p_i = 1/(s_{max} - s_{min} + 1)$ for $i = s_{min}, \ldots, s_{max}$ and $p_i = 0$ for $i = 1, \ldots, s_{min} - 1, s_{max} + 1, \ldots, S$. So, the distribution of size of the items is uniform only in expectation and there can be ample variations in the minimum number of bins required by different problem instances.

and/or for which there are many gaps in the bins. One can see these strategies in action during the packing of 500 items of sizes uniformly distributed in the range [21,70] in the last two columns of Figure 1.

We compare the performance of the hand-designed strategies and of a selection of 13 evolved strategies in the upper part of Table IV, where we report the average of the number of bins used by each strategy over the classes of problems from which testing instances in the fitness function were drawn. The column labelled "THEOR" represents the average minimum number of bins theoretically required by the problem instances.

As expected, BFD (see column labelled "LAST") is extremely good when the distribution of sizes is uniform within the range $[1, 150]$ – the type of range over which bin-packing algorithms are typically analysed. Also, BFD is very good in the range [1,80], although almost all algorithms did very well in this range. However, BFD loses badly its lead in the ranges [20,80] and [30,70] where it is outperformed by most of the evolved algorithms. A better all-rounder among the manually designed algorithms is SMIN.

To test the generality of hand-designed and evolved algorithms, we performed a further series of tests with different item distributions and bin sizes. The results are shown in the lower part of Table IV. These confirm that all evolved rules generalise well. Again, BFD/LAST does best when the distribution of sizes is uniform and spans ranges of the form $[1, s_{max}]$, but the opposite is observed in other situations.

In order to analyse the variations in behaviour of all algorithms, we collected statistics concerning which algorithm performed better than which other algorithm when compared head to head on the same problem instances. For each testing condition (bin size, size range and number of items) these statistics take the form of an array A with as many rows and columns as the number of algorithms under test (17). Element $A(a_1, a_2)$ represents the number of times algorithm a_1 did better than the algorithm a_2 in a trial problem. Ties are counted as 1/2 of a win. Results are then divided by the number of trials (100), so entries represent estimates of the probability of an algorithm outperforming another.

Because we use 32 conditions (12 for training, 20 for testing), we constructed 32 such arrays. Naturally, we cannot report them here. So, we averaged the contents of each array by row (excluding the diagonal elements which are meaningless), obtaining the average probability $p(a)$ with which a particular algorithm can beat the others. That is

$$p(a) = \frac{1}{16} \sum_{a'} A(a, a')$$

The values of $p(a)$ are reported for all 32 conditions and for the 17 algorithms under test in Table V. At the bottom of the table we also report the performance of each algorithm averaged over the 12 training problem classes, the 20 test problem classes or the full set of 32. Note that average values higher than 0.5 mean that an algorithm tends to beat other algorithms more often than they beat it, and *vice versa*. Many algorithms have conditions where they win against most other competitors, and *vice versa* as shown by the reported standard deviations. Some algorithms, however, are less prone to such variations.

The results reveal that, while there are size distributions for which it is difficult to beat BFD, most GP evolved heuristics can beat it on the training set, on the test set, and overall. SMIN is harder to beat, but again there are several evolved heuristics that do so. It is interesting to note that such heuristics not only outperform BFD and SMIN, but they also are more robust, as indicated by their much lower standard deviations.

The overall champion regarding performance appears to be GP2, which after simplification is

$$f(s) = \frac{49}{4} \cdot \frac{s^2 (s_{min})^2}{S^4} + 10^{-4},$$

while GP13 is a champion for reliability. After simplification the GP13 heuristic is

$$f(s) = \frac{s_{max} + s_{min} + s}{S} + 10^{-4}.$$

Both are extremely simple, yet effective and reliable.

VI. Conclusions

We presented a novel algorithm for the one-dimension offline bin packing problem with discrete item sizes. The algorithm works by progressively reducing the distance between the item-size distribution and distribution of gaps available in bins. The approach is controlled by a two functions: one function which decides how to prioritise items in order to decide which one to insert next, the other function decides how to prioritise gaps in order of urgency. Acting within Algorithm 1, these two functions progressively reduce the difference between item and gap histograms.

In this work we have kept the gap-selection function constant, and varied the item-selection function. This has allowed us both to manually define and to evolve high-performing bin-packing heuristics. Evolution was performed by a simple form of linear register-based GP.

We tested our evolved heuristics and compared them with hand-designed ones, including the well-known BFD heuristic and SMIN, a highly competitive, human-designed histogram-matching heuristic. Results have been very encouraging with most of the evolved heuristics outperforming human-designed heuristics.

In the future we will look at whether joint evolution of the item-selection function with the gap-selection function may afford further performance improvements. In addition, we intend to extend the approach to two- and three-dimensional bin packing, which can be trivially done by using multi-dimensional item- and gap-size histograms, and evolving item-selection functions of the form $f(s_1, s_2, \ldots)$. Also, we intend to extend the approach to more traditional continuous bin-packing problems where item sizes are reals. This extension can be obtained by quantising gap and item sizes.

Acknowledgements

The authors would like to thank EPSRC (Grants EP/C523377/1 and EP/C523385/1) for financial support. Edward Tsang, Matt Hyde, Graham Kendall, Robert Keller and Mohamed Bader-El-Den are also thanked for their useful comments and suggestions.

TABLE III

SAMPLE EVOLVED HEURISTICS FOR HISTOGRAM-MATCHING

Name	*Code*	*Function* $f(s)$
GP-S1	A=A*R4 A=A+B A=A+B A=A*R1 A=A+R3	$\frac{(o_s+2)s+s_{min}}{S} + 10^{-4}$
GP-S2	A=A+R4 A=A+R5 A=sqrt(A) NOP A=A+R1 A=A*R2 NOP NOP NOP NOP	$\frac{\left(\sqrt{o_s+g_s+1}+\frac{s}{S}\right)s_{max}}{S} + 10^{-4}$

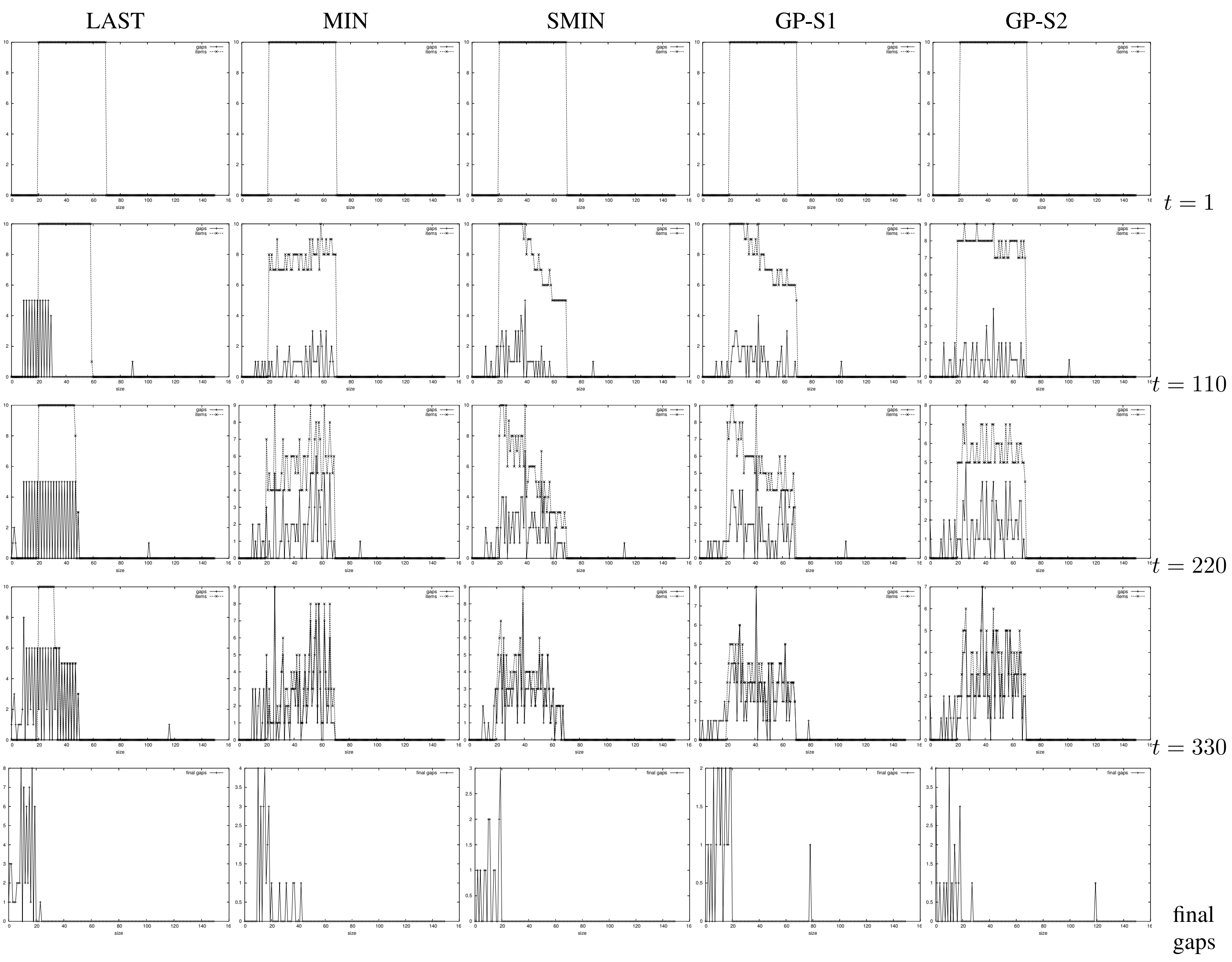

Fig. 1. Snapshots of $g_s(t)$ and $b_s(t)$ at different time steps and for different histogram matching algorithms.

REFERENCES

[1] E. Burke, M. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature (PPSN)*, 2006.

[2] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: an emerging direction in modern search technology, 2003.

[3] E. Coffman Jr, M. Garey, and D. Johnson. Approximation algorithms for bin packing: a survey. *Approximation algorithms for NP-hard problems table of contents*, pages 46–93, 1996.

[4] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing, 1996.

[5] K. Fleszar and K. S. Hindi. New heuristics for one-dimensional bin-packing. *Comput. Oper. Res.*, 29(7):821–839, 2002.

[6] H. J. Gupta JND. A new heuristic algorithm for the one-dimensional bin-packing problem. *Production Planning and Control*, 10(6):598–603, 1999.

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

[8] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[9] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Appl. Math.*, 28(1):59–70, 1990.

[10] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits - part i. *Genetic Programming and Evolvable Machines*, 1(1/2):7–35, 2000.

[11] P. Ross, J. Marin, S. Schulenburg, and E. Hart. Learning a procedure that can solve hard bin-packing problems: A new ga-based approach to hyperheuristics, 2003.

TABLE IV

PERFORMANCE OF HAND-DESIGNED (LAST/BFD, MIN, SMIN, RAND) AND EVOLVED (GP1–GP13) STRATEGIES ON THE CLASSES OF PROBLEMS FROM WHICH TESTING INSTANCES WERE DRAWN (UPPER 12 ROWS OF TABLE) AND ON AN INDEPENDENT TEST SET (LOWER 20 ROWS). FIGURES ARE MEANS OVER 100 INDEPENDENT TRIALS. THEOR IS THE MINIMUM NUMBER OF BINS THEORETICALLY REQUIRED BY THE PROBLEM INSTANCES.

S	Range	Items	THEOR	LAST	MIN	SMIN	RAND	GP1	GP2	GP3	GP4	GP5	GP6	GP7	GP8	GP9	GP10	GP11	GP12	GP13
150	[20,80]	100	33.89	34.66	36.4	34.68	35.32	34.59	34.65	34.61	34.74	34.58	34.63	34.63	34.62	34.61	34.6	34.63	34.7	34.64
		250	83.6	85.39	86.74	84.84	85.83	84.58	84.72	84.57	84.93	84.59	84.56	84.6	84.55	84.55	84.65	84.52	84.82	84.64
		1000	333.55	340.26	339.21	336.59	338.7	335.88	336.11	335.82	336.77	335.9	335.95	336.07	335.74	335.85	335.99	336.02	336.27	336.12
	[1,150]	100	51.02	53.21	58.88	53.3	55.12	53.36	53.29	53.38	53.31	53.33	53.41	53.32	53.43	53.38	53.3	53.41	53.31	53.35
		250	126.61	130.08	139.1	130.44	134.01	130.73	130.24	130.82	130.33	130.61	130.87	130.55	131.03	130.86	130.37	130.86	130.49	130.69
		1000	505.27	512.85	525.31	513.77	521.84	515.16	513.19	516.11	513.47	514.16	518.95	514.31	515.72	515.36	513.69	515.42	515.14	513.86
	[30,70]	100	33.85	36.23	36.31	35.35	36	35.49	35.4	35.51	35.41	35.54	35.45	35.41	35.52	35.47	35.45	35.44	35.42	35.43
		250	83.86	89.81	88.52	87.1	88.25	86.9	86.9	86.82	87.31	86.91	86.83	86.73	86.84	86.79	87.05	86.78	87.05	86.91
		1000	333.79	357.87	348.63	346.64	347.83	344.58	344.97	344.3	347.21	344.18	344.84	345.05	343.87	344.46	345.68	344.4	346.44	345.36
	[1,80]	100	27.43	27.48	29.36	27.46	28.01	27.48	27.49	27.48	27.48	27.51	27.48	27.49	27.48	27.49	27.47	27.48	27.48	27.5
		250	67.83	67.89	69.96	67.84	68.44	67.86	67.84	67.84	67.85	67.87	67.84	67.83	67.86	67.84	67.84	67.85	67.84	67.85
		1000	270.59	270.6	272.99	270.59	271.09	270.61	270.59	270.6	270.59	270.6	270.59	270.61	270.6	270.6	270.59	270.59	270.6	270.59
150	[20,80]	4000	1333.62	1360.57	1348.43	1343.26	1347.56	1342.32	1343.24	1342.11	1344.26	1341.98	1342.24	1342.58	1342.07	1341.84	1343.13	1342.28	1343.76	1342.79
	[1,150]	4000	2013.25	2028.5	2046.17	2030.33	2045.14	2032.3	2029.51	2034.04	2030.03	2030.71	2040.61	2031.03	2033.22	2032.55	2030.25	2032.91	2032.78	2030.49
	[30,70]	4000	1333.83	1429.65	1391.33	1383.86	1382.41	1378.59	1381.87	1376.93	1385.98	1376.65	1378.54	1380	1376.44	1378.04	1381.96	1378.65	1383.04	1380.23
	[1,80]	4000	1080.67	1080.67	1083.16	1080.67	1080.88	1080.67	1080.67	1080.68	1080.67	1080.67	1080.67	1080.67	1080.67	1080.67	1080.67	1080.67	1080.67	1080.67
75	[20,40]	100	40.28	43.71	43.75	43.94	44.47	43.86	44.07	43.58	44.02	43.92	43.68	44.15	43.58	43.76	43.9	43.71	44	43.87
		250	100.47	109.28	107.59	109.23	110.37	109.5	109.05	109.43	109.28	109.67	109.28	109.49	109.48	109.75	109.16	109.46	109.61	109.33
		1000	400.27	436.53	423.43	431.6	437.31	433.75	432.15	435.63	431.9	433.29	434.12	432.93	435.38	435.19	433.1	433.9	433.48	433.55
		4000	1600.25	1746.44	1687.03	1721.11	1743.22	1733.32	1723.27	1736.67	1723.51	1731.23	1736.99	1729.56	1735.16	1735.38	1727.78	1730.79	1729.27	1729.4
	[1,75]	100	50.91	52.98	56.38	53.12	54.6	53.27	52.99	53.27	53.1	53.16	53.28	53.13	53.29	53.29	53.1	53.28	53.13	53.15
		250	126.84	130.46	134.75	130.72	133.35	131	130.6	131.04	130.68	130.85	131.93	130.81	131.12	130.96	130.68	131.06	130.94	130.8
		1000	506.56	513.71	519.58	514.13	519.31	514.72	513.97	515.12	514.1	514.4	516.9	514.26	515	514.8	514.12	514.77	514.69	514.26
		4000	2026.69	2041.94	2051.28	2042.79	2052.83	2043.57	2042.42	2044.36	2042.71	2043.09	2049.33	2043.02	2044.03	2043.65	2042.84	2043.87	2044.16	2042.94
300	[20,160]	100	30.75	31	34.24	31.05	31.62	31.01	31.03	31.01	31.04	31.02	31.01	31.02	31.03	31.02	31.01	31.01	31.06	31.03
		250	75.6	76.02	80.17	76.07	76.93	75.92	75.97	75.97	76.1	75.95	76	75.92	75.95	75.96	75.96	76.01	76.02	76
		1000	300.23	301.51	305.53	300.91	302.49	300.71	300.7	300.66	300.95	300.7	300.67	300.8	300.66	300.67	300.8	300.69	300.85	300.71
		4000	1199.52	1203.85	1205.44	1201.19	1203.35	1200.78	1200.58	1200.73	1201.25	1200.76	1200.8	1200.93	1200.74	1200.67	1200.92	1200.81	1201.09	1200.74
	[1,300]	100	50.74	53.22	60.89	53.25	55.19	53.29	53.37	53.31	53.25	53.27	53.3	53.25	53.32	53.29	53.25	53.3	53.26	53.34
		250	126.09	129.81	144.44	130.06	134.39	130.28	130.28	130.35	129.97	130.24	130.53	130.07	130.63	130.46	130.06	130.53	130.07	130.45
		1000	501.08	508.62	532.46	509.99	520.58	511.61	509.29	512.49	509.68	510.93	512.69	510.82	512.63	512.22	509.91	512.34	510.66	510.94
		4000	2007.54	2022.79	2055.99	2026.42	2049.48	2031.02	2023.99	2034.48	2025.67	2027.27	2044.37	2027.79	2032.72	2031.48	2026.03	2032.09	2031.96	2026.81

TABLE V

HEAD TO HEAD COMPARISON BETWEEN HEURISTICS. FIGURES REPRESENT THE FRACTION OF TIMES AN ALGORITHM WON OVER ALL OTHERS IN 100 TEST PROBLEMS.

S	Range	Items	LAST	MIN	SMIN	RAND	GP1	GP2	GP3	GP4	GP5	GP6	GP7	GP8	GP9	GP10	GP11	GP12	GP13
150	[20,80]	100	0.5387	0.01094	0.5294	0.2381	0.5744	0.5444	0.5641	0.4997	0.5787	0.5541	0.5538	0.5587	0.5641	0.5691	0.5541	0.5194	0.5484
		250	0.2341	0.01469	0.4888	0.1422	0.6134	0.5472	0.6188	0.4453	0.6088	0.6225	0.6038	0.6281	0.6259	0.5794	0.6416	0.5003	0.5853
		1000	0.0175	0.07312	0.4025	0.1056	0.6694	0.5809	0.6887	0.3475	0.6606	0.6413	0.5928	0.7159	0.6775	0.6263	0.6137	0.5159	0.5706
	[1,150]	100	0.6159	0.0003125	0.5784	0.1078	0.55	0.5828	0.5422	0.5734	0.5641	0.5263	0.5687	0.5169	0.5406	0.5781	0.5263	0.5737	0.5544
		250	0.7609	0.000625	0.6159	0.07406	0.5153	0.6972	0.4816	0.6591	0.5434	0.4578	0.5784	0.4128	0.4688	0.6416	0.4681	0.5938	0.5306
		1000	0.8913	0.0175	0.7066	0.07562	0.4475	0.8278	0.3369	0.7603	0.6275	0.2213	0.5944	0.3375	0.4031	0.7241	0.3819	0.4587	0.6881
	[30,70]	100	0.1791	0.1559	0.6109	0.3109	0.5406	0.5863	0.5312	0.5813	0.515	0.5612	0.5809	0.5262	0.5497	0.5606	0.5666	0.5734	0.57
		250	0.009375	0.1028	0.5141	0.1584	0.5972	0.5953	0.6325	0.4269	0.595	0.6262	0.6675	0.6231	0.6434	0.5331	0.6481	0.5341	0.5928
		1000	0	0.1053	0.2953	0.2034	0.6903	0.61	0.7497	0.2334	0.7737	0.6369	0.6	0.8284	0.7166	0.4641	0.7309	0.3275	0.5344
	[1,80]	100	0.5487	0.003438	0.5587	0.2903	0.5487	0.5438	0.5487	0.5487	0.5338	0.5487	0.5438	0.5487	0.5438	0.5537	0.5487	0.5487	0.5388
		250	0.5269	0.00375	0.5516	0.2931	0.5416	0.5516	0.5516	0.5466	0.5369	0.5516	0.5563	0.5416	0.5513	0.5516	0.5466	0.5513	0.5466
		1000	0.5431	0.001563	0.5481	0.3266	0.5381	0.5481	0.5431	0.5481	0.5431	0.5481	0.5381	0.5431	0.5431	0.5481	0.5481	0.5431	0.5481
150	[20,80]	4000	0	0.08281	0.4597	0.1091	0.6741	0.4594	0.725	0.285	0.7528	0.6975	0.6175	0.7328	0.7928	0.4928	0.6884	0.365	0.5653
	[1,150]	4000	0.93	0.04094	0.7044	0.05969	0.465	0.8363	0.3266	0.7547	0.6569	0.1475	0.6247	0.3437	0.4238	0.7147	0.3884	0.4128	0.67
	[30,70]	4000	0	0.06437	0.2628	0.3709	0.7009	0.4084	0.8169	0.1675	0.8575	0.7053	0.5653	0.8794	0.7397	0.3919	0.6997	0.3206	0.5487
	[1,80]	4000	0.5381	0	0.5381	0.4331	0.5381	0.5381	0.5331	0.5381	0.5381	0.5381	0.5381	0.5381	0.5381	0.5381	0.5381	0.5381	0.5381
75	[20,40]	100	0.5709	0.5491	0.4753	0.265	0.505	0.4278	0.6341	0.435	0.4766	0.5903	0.3681	0.6331	0.5487	0.4962	0.5781	0.4403	0.5062
		250	0.5247	0.8872	0.5569	0.235	0.4641	0.5984	0.4759	0.5366	0.4047	0.5363	0.4528	0.4669	0.3853	0.5703	0.4778	0.42	0.5072
		1000	0.1616	1	0.8	0.09781	0.4941	0.7119	0.235	0.7394	0.5656	0.4316	0.625	0.2531	0.28	0.5856	0.4666	0.5272	0.5256
		4000	0.02219	1	0.9009	0.06687	0.3903	0.8469	0.2538	0.8322	0.4803	0.2103	0.5697	0.2809	0.2756	0.6684	0.5209	0.6028	0.5778
	[1,75]	100	0.6416	0.009063	0.58	0.1306	0.5106	0.6372	0.51	0.5891	0.5619	0.5084	0.5756	0.5019	0.5034	0.5897	0.5088	0.5753	0.5669
		250	0.7222	0.02906	0.6166	0.09906	0.5138	0.6647	0.4944	0.6331	0.5684	0.3488	0.5831	0.4697	0.5228	0.6306	0.4875	0.5337	0.5825
		1000	0.7872	0.04813	0.6669	0.05281	0.4956	0.7141	0.4134	0.6766	0.5881	0.2425	0.6306	0.4125	0.4688	0.6744	0.4847	0.5156	0.6281
		4000	0.8228	0.07812	0.6772	0.04	0.5166	0.745	0.3831	0.6913	0.615	0.1913	0.6294	0.4266	0.5003	0.6544	0.4591	0.4166	0.6534
300	[20,160]	100	0.5612	0	0.5359	0.2575	0.5566	0.5463	0.5562	0.5416	0.5513	0.5556	0.5513	0.5463	0.5509	0.5559	0.5556	0.5316	0.5463
		250	0.5425	0.0003125	0.5175	0.1322	0.5909	0.5669	0.5669	0.5025	0.5766	0.5519	0.5909	0.5766	0.5719	0.5716	0.5469	0.5419	0.5522
		1000	0.2328	0	0.5094	0.07156	0.6031	0.6084	0.6266	0.4906	0.6081	0.6219	0.5616	0.6272	0.6219	0.5619	0.6131	0.5384	0.6034
		4000	0.07719	0.00625	0.45	0.1044	0.6266	0.7137	0.6488	0.4272	0.6347	0.6181	0.5613	0.6434	0.6744	0.5659	0.6134	0.4934	0.6412
	[1,300]	100	0.5922	0	0.5781	0.07719	0.5587	0.5209	0.5494	0.5781	0.5687	0.5544	0.5778	0.5447	0.5591	0.5781	0.5544	0.5731	0.535
		250	0.7241	0	0.6291	0.06344	0.5534	0.5534	0.5269	0.6656	0.5647	0.4641	0.6269	0.4356	0.4863	0.6266	0.4641	0.6262	0.4897
		1000	0.9472	0	0.7163	0.0625	0.4581	0.8378	0.3563	0.7684	0.5706	0.2959	0.5697	0.2888	0.3562	0.73	0.345	0.6025	0.5947
		4000	0.9731	0.02781	0.7188	0.07312	0.4316	0.91	0.2609	0.8072	0.6506	0.1297	0.6197	0.3109	0.3987	0.7603	0.3572	0.3787	0.6916
Training	set	Mean	0.4055	0.04083	0.5334	0.1938	0.5689	0.6013	0.5658	0.5142	0.5901	0.5413	0.5815	0.5651	0.569	0.5775	0.5646	0.52	0.5673
		Std	0.3062	0.05391	0.1054	0.09533	0.06642	0.08332	0.1038	0.1389	0.07217	0.1141	0.03492	0.1278	0.08672	0.06415	0.08949	0.07067	0.0428
Test	set	Mean	0.5186	0.1912	0.5947	0.1401	0.5324	0.6423	0.4947	0.583	0.5896	0.447	0.572	0.4956	0.5099	0.5979	0.5174	0.4977	0.5762
		Std	0.3262	0.354	0.1409	0.1119	0.07855	0.1453	0.1588	0.1715	0.09627	0.1851	0.06401	0.1617	0.1361	0.08794	0.09546	0.08748	0.05685
Overall		Mean	0.4762	0.1348	0.5717	0.1602	0.5461	0.6269	0.5213	0.5572	0.5897	0.4824	0.5756	0.5217	0.5321	0.5902	0.5351	0.5061	0.5729
		Std	0.3187	0.2886	0.1305	0.1077	0.0753	0.1257	0.1432	0.1613	0.08676	0.1667	0.05446	0.1516	0.1219	0.07938	0.09468	0.08113	0.05147