Learning a procedure that can solve hard bin-packing problems: a new GA-based approach to hyper-heuristics

No Author Given

No Institute Given

Abstract. The idea underlying hyper-heuristics is to discover some combination of familiar, straightforward heuristics that performs very well across a whole range of problems. To be worthwhile, such a combination should outperform all of the constituent heuristics. In this paper we describe a novel messy-GA-based approach that learns such a heuristic combination for solving one-dimensional bin-packing problems. When applied to a large set of benchmark problems, the learned procedure finds an optimal solution for nearly 80% of them, and for the rest produces an answer very close to optimal. When compared with its own constituent heuristics, it ranks first in 98% of the problems.

1 Introduction

A frequent criticism of evolutionary algorithms (EAs) is that the solutions they find can be fragile. Although the EA 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. For this reason, users may prefer to stick with using familiar and simpler but less effective heuristics. The idea of *hyper-heuristics* is to use a search process such as an EA, not to solve a given problem but to discover a combination of straightforward heuristics that can perform well on a whole range of problems.

In this paper we consider one-dimensional bin-packing problems, described below. These are simple to understand but the class is NP-hard, and they are worth studying because they appear as a factor in many other kinds of optimization problem. A very simplified representation of the possible state of any problem is used, and a messy GA is used to associate a heuristic with various specific instances of this representation. The problem solution procedure is Given a state P of the problem, find the nearest instance I and apply the associated heuristic $H(I)$. This transforms the problem state, say to P'. Repeat this until the problem has been solved.

The task of the GA is to choose the problem-state instances and to associate one of a small set of heuristics with each of them. Any given heuristic might be associated with several instances, or with none.

An example of a hyper-heuristic approach applied to the one-dimensional bin-packing problem has been reported in [12] and [13], where an accuracy-based Learning Classifier System (XCS) [14] was used to learn a set of rules that

associated characteristics of the current state of a problem with eight different algorithm heuristics. In this previous work a classifier system was trained on a set of example problems and showed good generalisation to unseen problems. This represented a useful step towards the concept of using EAs to generate strong solution processes rather than merely using them to find good individual solutions. However, some questions arose from that work: could performance be improved further, and was a classifier system necessary to the approach? XCS focuses rewards on single actions [12] or on short groups of actions [13] – what if rewards are given only by the final outcome?

This paper answers these questions. It presents an alternative approach tested using a large set of benchmark one-dimensional bin-packing problems and a small set of eight heuristics. No single one of the heuristics used is capable of finding the optimal solution of more than a very few of the problems; however, the evolved rule-set was able to produce an optimal solution for nearly 80% of them, and in the rest it produced a solution very close to optimal. When compared with the solutions given by its own constituent heuristics, it ranks first in 98%.

2 One-dimensional Bin-packing

In the one-dimensional bin packing problem, there is an unlimited supply of bins, each with capacity $c > 0$. A set of n items is to be packed into the bins, the size of item i is $s_i > 0$, and:

$$\forall k : \sum_{i \in \text{bin}(k)} s_i \leq c \tag{1}$$

The task is to minimise the total number of bins used. Despite its simplicity, this is an NP-hard problem. If M is the minimal number of bins needed, then clearly:

$$M \geq \lceil (\sum_{i=1}^{n} s_i)/c \rceil \tag{2}$$

and for any algorithm that does not start new bins unnecessarily, $M \leq$ bins used $< 2M$ because otherwise there would be two bins less than half filled.

Many results are known about specific algorithms. For example, a commonly-used algorithm is Largest-Fit-Decreasing (LFD): items are taken in order of size, largest first, and put in the first bin where they will fit (a new bin is opened if necessary, and effectively all bins stay open). It is known [9] that this uses no more than $11M/9 + 4$ bins. A good survey of such results can be found in [3]. A good introduction to bin-packing algorithms can be found in [11], which also introduced a widely-used heuristic algorithm, the Martello-Toth Reduction Procedure (MRTP). This simply tries to repeatedly reduce the problem to a simpler one by finding a combination of 1-3 items that provably does better than anything else (not just any combination of 1-3 items) at filling a bin, and if so packing them. This may halt with some items still unpacked; the remainder are packed using LFD. The reader may wonder whether it pays to try to fill each bin as much as possible, but [12] shows a simple counter-example.

Various authors have applied EAs to bin-packing, notably Falkenauer's grouping GA [5, 7, 6]; see also [10] for a different approach. Falkenauer also produced two of several sets of benchmark problems. In one of these, the so called *triplet problems*, every bin contains three items; they were generated by first constructing a solution which filled every bin exactly, and then randomly shrinking items a little so that the total shrinkage was less than the bin capacity so that the number of bins needed is unaffected. [8] raised a question about whether these problems are genuinely hard or not. It is certainly possible to solve many of them very quickly by backtracking if you exploit the knowledge that there is an optimal solution in which every bin contains three items.

3 Bin-Packing Benchmark Problems

Problems from several sources have been used in this investigation. One collection, available from Beasley's OR-Library [1], contains problems of two kinds that were generated and largely studied by Falkenauer [6]. The first kind, 80 problems named `uN_M`, involve bins of capacity 150. N items are generated with sizes chosen randomly from the interval 20-100. For each N in the set $(120, 250, 500, 1000)$ there are twenty problems, thus M ranges from 00 to 19. The second kind, 80 problems named `tN_M`, are the triplet problems mentioned earlier. The bins have capacity 1000. The number of items N is one of 60, 120, 249, 501 (all divisible by three), and as before there are twenty problems per value of N. Item sizes range from 250 to 499; the problem generation process was described earlier.

A second collection of problems studied in this paper comes from the Operational Research Library [2] at the *Technische Universität Darmstadt*: we use the 'bpp1-1' and the very hard 'bpp1-3' sets in this paper. In the bpp1-1 set problems are named `NxCyWz_a` where `x` is 1 (50 items), 2 (100 items), 3 (200 items) or 4 (500 items); `y` is 1 (capacity 100), 2 (capacity 120) or 3 (capacity 150); `z` is 1 (sizes in 1*ldotds*100), 2 (sizes in 20...100) or 4 (sizes in 30...100); and `a` is a letter in A... T indexing the twenty problems per parameter set. (Martello and Toth [11] also used a set with sizes drawn from 50...100, but these are far too easy.) In the hard bpp1-3 set there are just ten problems, each with 200 items and bin capacity 100,000; item sizes are drawn from the range $20,000...35,000$.

Finally, a group of random problems has also been created in an attempt to be as fair as possible. The above benchmark problems do not use small item sizes, because small items could be used as sand (so to speak) to fill up boxes. A procedure that works well on such benchmark problems might still perform badly on problems that include small items. We generate some problems in which the bins have capacity of 100 or 500, and there are 100, 250 or 500 items whose sizes are chosen at uniform random from 1 up to the bin capacity. In all, therefore, 1016 benchmark problems are used. These were divided in two groups, a training set with 763 problems and a test set with 253.

4 The set of heuristics used

Following [12], we used four basic heuristics:

- LFD, described in Section 2 above. This was the best of the fourteen heuristics in over 81% of the bpp1-1 problems, but was never the winner in the bpp1-3 problems.
- Next-Fit-Decreasing (NFD): an item is placed in the current bin if possible, or else a new bin is opened and becomes the current bin and the item is put in there. This is usually very poor.
- Djang and Finch's algorithm (DJD) [4]. This puts items into a bin, taking items largest-first, until that bin is at least one third full. It then tries to find one, or two, or three items that completely fill the bin. If there is no such combination it tries again, but looking instead for a combination that fills the bin to within 1 of its capacity. If that fails, it tries to find such a combination that fills the bin to within 2 of its capacity; and so on. This of course gets excellent results on, for example, Falkenauer's problems; it was the best performer on just over 79% of those problems but was never the winner on the hard bpp1-3 problems.
- DJT (Djang and Finch, more tuples): a modified form of DJD considering combinations of up to five items rather than three items. In the Falkenauer problems, DJT performs exactly like DJD, as could be expected because of the way Falkenauer problems are generated; in the bpp1-1 problems it is a little better than DJD.

As in [12], to four more heuristics, these algorithms were each coupled with a 'filler' process that tried to find any item at all to pack in any open bins rather than moving on to a new bin. This might, for example, make a difference in DJD if a bin could be better filled by using more than three items once the bin was one-third full. So, for example, the heuristic 'Filler+LFD' first tries to fill any open bins as much as possible — this process terminates if the filling action successfully inserts at least one item. If no insertion was possible, 'Filler+LFD' invokes LFD. This forces progress: at least one gets put in a bin. Without this, endless looping might be possible. Thus, in all eight heuristics have been used. For convenience they are numbered 0..7.

5 Representing the problem state

The problem state is defined by five real numbers. The first four numbers give the ratio R of items remaining to be packed that fall into each of the four categories listed in table 1. These intervals are, in a sense, natural choices since at most one huge item will fit in a bin, at most two large items will fit a bin, and so on. The fifth number represents the fraction of the original number of items that remain to be packed; in a sense, it indicates the degree of progress through the problem.

Table 1. Item size ranges: c = bin capacity

Huge:	items with $c/2 < s_i$
Large:	items with $c/3 < s_i \leq c/2$
Medium:	items with $c/4 < s_i \leq c/3$
Small:	items with $s_i \leq c/4$

It is worth noting that this representation throws away a lot of detail. For each of the benchmark problems, we tested which individual heuristics were winners on that problem. It turned out that the task of associating the initial problem state with the winning heuristics was not a linearly separable one.

6 The Genetic Algorithm

6.1 Representation

A chromosome is composed of blocks, and each block j contains six numbers $h_j, l_j, m_j, s_j, i_j, a_j$. The first five essentially represent an instance of the problem state. Here h_j corresponds to the proportion of huge items that remain to be packed, and similarly l_j, m_j and s_j refer to large medium and small items, and i_j corresponds to the proportion of items remaining to be packed. The sixth number, a_j, is an integer in the range $0 \cdots 7$ indicating which heuristic is associated with this instance. An example of a set of 12 rules obtained with the GA can be seen in figure 1.

Fig. 1. Example of a final set with 12 rules

```
0.70 -2.16 -1.10  1.55  1.81 --> 1  |   2.34  0.67  0.19  1.93  2.75 --> 1
0.12  1.37 -0.54  1.12  0.58 --> 6  |  -1.93 -2.64 -1.89  2.17 -1.46 --> 3
0.13  1.43 -1.27  0.13 -2.18 --> 2  |  -1.30  0.11  2.00 -1.85  0.84 --> 4
1.87 -0.91  1.30 -1.34  1.93 --> 3  |   0.32  1.94  2.24  0.99 -0.53 --> 0
2.60  1.30 -0.54  1.12  0.58 --> 6  |   0.58  0.87  0.23 -2.11  0.47 --> 1
0.25  2.09 -1.50 -1.46 -2.56 --> 0  |   1.21  0.11  2.00  0.09  0.84 --> 4
```

For a given problem, the first five numbers would all lie in the range 0.0 to 1.0; that is, the actual problem state is a point inside a five-dimensional unit cube. The blocks in the chromosome represent a number of points, and at each step the solution process applies the heuristic associated with the point in the chromosome that is closest to the actual problem state. However, we permit the points defined in the chromosome to lie outside the unit cube, by allowing the first five numbers in a block to be in the range $-2 \cdots 3$. This means that if, say, the initial problem state is $(0, 0, 0, 1, 1)$ (that is, all items are small), the nearest instance is not compelled to be an interior point of the unit cube.

It is unclear at the outset how many blocks are needed. It would be possible to use a fixed-length chromosome by guessing an upper limit on the number of

blocks and, for each block, having a bit to say whether that block is to be used or not. However, we use a variable-length chromosome instead.

6.2 The operators

We wish to keep things simple, to avoid any accusation that success rises from excessive complexity of approach. So, two very simple crossovers and three mutations have been implemented.

The first crossover is a standard two point crossover modified to work with a variable length representation and to avoid changing the meaning of any numbers. For any fixed-length GA, two point crossover uses the same two cross points for both parents and thus maintains the length of the child chromosomes. But here, each parent may choose the interchange points independently, with the caveat that the exchange points should fall inside a block in the same position for both parents. This ensures that at the end of the exchange process, the blocks are complete. The operator is implemented by choosing for each parent, a block in which to start the exchange (potentially different in each parent) and then an offset inside the block (the same in both parents). The same is done for the end point of exchange. The blocks and the points are chosen using a uniform distribution. The genetic material between these block points is then exchanged in the children.

The second crossover works at block level. It exchanges 10% of blocks between parents, that is, each block of let us say, the first parent, has a 90% chance of being passed to the first child and a 10% of being passed to the second child and vice-versa. Thus this operator merely shuffles blocks. Each crossover has an equal probability of being chosen.

The three mutations are: an add-block mutation, a remove-block mutation and a normal-mutation. The first one generates a new block and adds it to the chromosome. The first five numbers of a block are selected from a normal distribution with mean 0.5, and truncated to lie in $-2 \cdots 3$; the sixth number (the heuristic, an integer in $0 \cdots 7$) is chosen at uniform random. The remove-block mutation removes a random block from the chromosome. The normal-mutation changes a locus (at a rate of about one per four blocks) either adding a normally distributed random number in the case of the real values or, in the case of the heuristic, by choosing a value at uniform random.

A child has a 10% chance of being mutated. If it is to be mutated, then one type is chosen: add-block and remove-block each have a 25% chance of occurring, and normal-mutation has a 50% chance of occurring. These values were suggested by some early testing.

6.3 The fitness function

The fundamental ingredient in fitness is how well a chromosome solves a specified problem. But how is this to be measured? We did consider a ranking process, comparing the number of bins used with the best result so far for that problem. A chromosome's fitness would then be a weighted average of the ranking it

achieved (compared to any other chromosome's result) on each of the problems it had seen so far; the weighting would be proportional to the number of time that any chromosome had tackled that problem. Being the best of only three attempts to solve a problem is not worth much.

In the end we opted for a simpler, cheaper measure: how much better was the chromosome's result on the specified problem than any single heuristic had been able to achieve (latter referred in the tables as best of four heuristics or BFH). The single-heuristic results could be prepared in advance of running the GA.

We want a chromosome to be good at solving many problems. When a chromosome is created, it is applied to just five randomly-chosen problems in order to obtain an initial fitness. Then, at each cycle of the algorithm, a new problem is given to each of the chromosomes to solve. This procedure is a trade-off between accuracy and speed; newly-created chromosomes may have to tackle many problems but they are at least built from chromosomes that had experienced many of those problems before.

The longer a chromosome survives in a population the more accurate its fitness becomes, which raises the question of how to rate a young chromosome (with therefore a poor estimate) against an older chromosome (with an improved estimate) if the fitness of the younger one is better? Several methods were tried to attempt to give higher credit to older and therefore more accurate chromosomes:

- Fixed Weighted Sum: The age and the excess/rank are combined (after normalization) with fixed weights, age weight being rather low.
- Variable Weight Sum: The weights change with the progress, age being more and more important but always using relatively low values.
- Age Projection: The search keeps track of the average degradation with age and, when comparing a chromosome with an older chromosome the fitness is updated with the expected degradation given how good is the young for its age.
- Improved Age Projection: The degradation is decomposed year by year and the updating is incremental for each year of difference.

However, preliminary trials rather surprisingly indicated that age of chromosome was not an important factor; we discarded these attempts to make any special concessions to chromosome youngsters.

6.4 The genetic engine

The GA uses a steady-state paradigm. From a population two parents are chosen: one by tournament size 2, the other randomly. From these two parents two children are created. The children are given a subset of four training problems to solve and obtain an initial fitness. The two children replace (if better) the two worst members of the population. Then the whole population solves a new training problem (randomly selected for each chromosome) and fitnesses are updated. The process is repeated a fixed number of times.

7 Experimental details

The GA described above was run five times with different seeds and the results averaged. The size of the population is 40, the GA is run for 500 iterations.

That gives 24160 problems solved, 40 problems (one each member of the population) plus 2 children times 4 (initialization) problems each generation plus 40 times 4 problems for initial initialization. That means that, on average, each training problem has been seen about 31 times.

For training, all available bin-packing problems were divided into a training and a test set. In each case the training set contained 75% of the problems; every fourth problem was placed in the test set. Since the problems come in groups of twenty for each set of parameters, the different sorts of problem were well represented in both training and test sets.

At the end of training, the fittest chromosome is used on every problem in the training set to assess how well it works. It is also applied to every problem in the test set.

The initial chromosomes are composed of 16 randomly created blocks. It is of course possible to inject a chromosome that represents a single heuristic simply by specifying that heuristic in every block, and so test how your favourite heuristic fares compared to evolved hyper-heuristics. It can be argued that the imprecision regarding the fitness can kill these individual heuristics but they would have first to be the worst of the population to be in a situation were they could be removed. Being the worst of a population, even with imprecise estimations of quality, hardly qualifies as a potential winner.

8 Results

Tables 2 and 3 show accumulative percentages of problems solved with a particular number of extra bins when compared with the results of, respectively, the best of the four heuristics (BFH) and the literature reported optimal value. HH-Methods stand for Hyper-heuristic methods, XCSs and XCSm for the results of the XCS in its single [12] and multi-step [13] versions. Finally, Trn and Tst stand for results obtained with the training and testing set of problems respectively.

Accumulative percentages means that a cell is read as percentage of problems solved with x **or less** bins than the comparison reference (BFH or reported).

Although the main focus and achievement of this work is to be able to improve results by finding smart combinations of individual heuristics, it also seems interesting to see the performance versus known optimal results (even if the heuristics to be combined can not achieve them). Therefore table 3 is also presented. Not all problems used to train and test the GA (and the other methods) have reported optimal values and therefore table 3 uses a smaller number of problems, in particular the bpp1-1, bpp1-3 and Falkenauer problems.

Table 2 shows that there is little room for improvement over the best of the four heuristics (DJT being the best in about 95% of the problems) but the HH-Methods have achieved better performance. GA achieves about 98% of the

Table 2. Extra bins compared to best of four heuristics (BFH)

	HH Methods						Heuristics							
	GA		XCSs		XCSm		LFD		NFD		DJD		DJT	
Bins	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst
-4		0.4												
-3	0.3	0.8												
-2	1.3	1.2	0.3	0.5	0.3	0.9								
-1	4.2	5.5	2.7	2.2	2.3	3.6								
0	98.3	97.6	98.3	97.3	98.8	97.3	71.1	73.9			91.2	91.7	95.4	94.1
1	100	100	100	100	100	100	83.8	82.6	0.1		97.3	97.6	99.7	99.6
2							88.9	88.5	0.1		98	98.4	100	100
3							91.9	92.5	1.1	2	99.6	98.8		
4							93.8	93.3	3.7	4	100	99.6		
5							95.8	96.1	7.2	5.9		100		
10							97.4	96.8	25.3	24.5				
20							99.7	99.6	48.1	47.8				
30							100	100	61.1	60.5				

Table 3. Extra bins compared versus optimal reported

	HH Methods						Heuristics							
	GA		XCSs		XCSm		LFD		NFD		DJD		DJT	
Bins	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst	Trn	Tst
0	78.8	78.8	78.7	76.2	79	76.7	61.5	64.4			70	69.4	74.3	71.6
1	96	95.4	94.9	94.6	94	94.6	78.1	77.9	0.2		90.7	91.4	93.6	94.1
2	98.7	99.4	97.6	97.8	97.6	98.2	82.5	82	0.2		94.9	95.5	97.3	97.3
3	100	100	98.8	98.7	98.8	98.7	88.6	88.7	0.3	0.9	98	97.3	98.8	98.7
4			99.4	99.1	99.4	99.1	91.6	91.9	2.8	3.6	99.4	98.7	99.4	99.1
5			100	99.5	100	99.5	93.7	92.3	7.9	6.3	100	99.5	100	99.5
6				99.5		99.5	95.2	95.5	12.2	11.3		99.5		99.5
7				100		100	95.5	95.5	16.7	16.2		100		100
10							95.5	95.5	27.5	27				
20							97.8	97.8	50.2	50				
30							100	100	61.5	61.7				

problems with 0 or less bins (getting a 3% improvement). Even more, in about 5% of cases it outperforms its own constituent heuristics by packing all items in less bins than the best single heuristic, with an interesting maximum improvement of 4 bins less in 1% of the cases. XCS's overall performance is much the same than the GA, however, the percentage of saved bins (negative bins) is smaller than the GA, just about 2.5%. But to be fair, it is important to remember that XCS works on binary state representations with only 2 bits to encode the range $[0 \ldots 1]$.

As table 3 shows, while DJT reaches optimality in 73% of the cases, the HH methods achieve up to 79%. It is important to note the excellent performance of the GA that, at worst, would use three extra bins (and just in 1% of the cases) while the other methods have worst cases of seven extra bins. This shows the robustness that hyper-heuristics can achieve.

In addition, the GA is able to generalise well. Results with training problems are very close to results using the new test cases. This means that particular details learned (a structure of some kind) during the adaptive phase can be reproduced with completely new data (unseen problems taken from the test sets).

Table 4 shows the distribution of actions performed when all the problems are solved using the hyper-heuristic methods. The averaged column refers to the distributions taking all the different sets of rules obtained using several seeds. The single best rule set column shows the distribution for just one set of rules, in particular the one that gives the best results.

Table 4. Distribution of actions performed when solving problems

		Averaged			Single best rule set		
Action	Heur	GA	XCSs	XCSm	GA	XCSs	XCSm
0	LFD	0.1	3.09	14.20		29.18	
1	NFD		0.37				
2	DJD	15	17.47	22.66		24.46	18.67
3	DJT	3.7	25.39	20.76		4.58	45.05
4	LFD+F	4.3	18.46	5.949	7.02		10.91
5	NFD+F	1.7					
6	DJD+F	33.4	18.42	20.07	92.98	38.87	25.37
7	DJT+F	41.9	16.80	16.35		2.91	

Heuristic NFD has a very poor performance as tables 2 and 3 demonstrate (never reaching being the best of the four heuristic in any presented problem) and the HH methods learn that fact (very low proportion of actions of that heuristic). However, very interestingly, sometimes it is useful as it is executed, in up to 2% of the actions, by rules obtained by the GA (although not in the best sets of rules) that is the winner of all bin packing methods.

XCS learns to use DJs heuristics in about 80% of the cases and LFD the other 20% of the time, with an even distribution between using or not using the filler. On the other hand, the GA learns to use DJs in about 95% of the cases and clearly favours the use of the filler. The examples of particular sets of rules seems to fit more or less into these distributions of actions.

It has to be stated that the resultant sets of rules were **never** composed of a single heuristic and that no chromosome with one block only survived in any of the tested populations. This also happened in the experiments done to tune the GA. This supports that Hyper-heuristics seems then to be the path to follow.

9 Conclusions

This paper represents another step towards developing the concept of hyper-heuristics: using EAs to find powerful combinations of more familiar heuristics.

From the experiments shown it is also interesting to note that:

1. It was shown in [12, 13] that XCS is able to create and develop feasible hyper-heuristics that performed well on a large collection of benchmark data sets found in literature, and better than any individual heuristic. This work consolidates such conclusion from another approach, using a GA, and using a different reward schema.
2. The system always performed better than the best of the algorithms involved, and in fact produced results that were either optimal (in the large majority of cases) or, at most, and in just 1.5% of the cases, one extra bin.
3. In about 5% of the cases the hyper-heuristics are able to find a combination of heuristics that **improve** the individual application of each of them.
4. The worst case of the GA compared versus the reported optimal value uses three extra bins (and just in 1% of the problems) while the other methods have worst cases of seven extra bins.

The current rules can be difficult to interpret (see figure 1). The distance between the antecedents and the problem state is a measure of how close are the concepts they represent. Long distances mean that the concepts are rather dissimilar. In the current implementation only being closer than any other rule is taken into account. No attempt to reduce as much as possible the average distance between the rules and the problem states is performed. Introducing a bias towards sets of rules with reduced distances would make rules much more interpretable.

On the other hand, among the particular advantages of the use of the GA it is interesting to mention that:

1. A particular set of rules can be tested for optimality by generating a chromosome containing them and injecting it into the population. This feature was used to show that individual heuristics were not optimal.
2. Variable chromosome length and encoding the action allows for a heuristic to be used in as many different contexts as necessary.
3. The use of real encoding allows for better precision to define the contexts where an action is preferred than if binary encoding of the problem state where used.
4. Rewards are given by final results. This allows for learning long chain of actions that locally can be seen as suboptimal but globally performs very well (like opening and underfilling some bins to fill them in latter stages). XCSs, for example, focuses in locally optimal actions (an action is rewarded regarding how well it filled a bin).
5. Fitness functions allows for other objectives to be taken into account (as compact sets of rules, minimized distances between rules antecedents and states, etc.).

In summary, HH methods performance is substantially better than the best heuristics alone with the GA being slightly better than the XCS method.

Acknowledgments

This work has been supported by UK EPSRC research grant number GR/N36660.

References

1. `http://www.ms.ic.ac.uk/info.html`.
2. `http://www.bwl.tu-darmstadt.de/bwl3/forsch/projekte/binpp/`.
3. E.G Coffman, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing, Boston, 1996.
4. Philipp A. Djang and Paul R. Finch. Solving One Dimensional Bin Packing Problems. 1998.
5. Emanuel Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary Computation*, 2(2):123–144, 1994.
6. Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996.
7. Emanuele Falkenauer. A Hybrid Grouping Genetic Algorithm for Bin Packing. Working Paper IDSIA-06-99, CRIF Industrial Management and Automation, CP 106 - P4, 50 av. F.D.Roosevelt, B-1050 Brussels, Belgium, 1994.
8. I. P. Gent. Heuristic Solution of Open Bin Packing Problems. *Journal of Heuristics*, 3(4):299–304, 1998.
9. D.S. Johnson. *Near-optimal bin-packing algorithms*. PhD thesis, MIT Department of Mathematics, 1973.
10. Sami Khuri, Martin Schutz, and Jörg Heitkötter. Evolutionary heuristics for the bin packing problem. In D. W. Pearson, N. C. Steele, , and R. F. Albrecht, editors, *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Ales, France, 1995*, 1995.
11. Silvano Martello and Paolo Toth. *Knapsack Problems. Algorithms and Computer Implementations*. John Wiley & Sons, 1990.
12. Peter Ross, Sonia Schulenburg, Javier G. Marín-Blázquez, and Emma Hart. Hyper-heuristics: learning to combine simple heuristics in bin packing problems. In *Genetic and Evolutionary Computation Conference*, New York, NY, 2002. Winner of the Best Paper Award in the Learning Classifier Systems Category of GECCO 2002.
13. Sonia Schulenburg, Peter Ross, Javier G. Marín-Blázquez, and Emma Hart. A Hyper-Heuristic Approach to Single and Multiple Step Environments in Bin-Packing Problems. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems: From Foundations to Applications*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2003. In press.
14. Stewart W. Wilson. Classifier Systems Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.