Hyper-heuristics: learning to combine simple heuristics in bin-packing problems

Peter Ross
School of Computing
Napier University
Edinburgh EH10 5DT
peter@dcs.napier.ac.uk

Sonia Schulenburg
School of Computing
Napier University
Edinburgh EH10 5DT
s.schulenburg@napier.ac.uk

Javier G. Marín-Blázquez
Division of Informatics,
The University of Edinburgh
Edinburgh EH1 1HN, UK
javierg@dai.ed.ac.uk

Emma Hart
School of Computing
Napier University
Edinburgh EH10 5DT
emmah@dcs.napier.ac.uk

Abstract

Evolutionary algorithms (EAs) often appear to be a 'black box', neither offering worst-case bounds nor any guarantee of optimality when used to solve individual problems. They can also take much longer than non-evolutionary methods. We try to address these concerns by using an EA, in particular the learning classi£er system XCS, to learn a solution process rather than to solve individual problems. The process chooses one of various simple non-evolutionary heuristics to apply to each state of a problem, gradually transforming the problem from its initial state to a solved state. We test this on a large set of one-dimensional bin packing problems. For some of the problems, none of the heuristics used can £nd an optimal answer; however, the evolved solution process can £nd an optimal solution in over 78% of cases.

1 INTRODUCTION

Heuristic algorithms are very widely used to tackle practical problems in operations research, because so many are NP-hard [12] and exhaustive search is often computationally intractable. Evolutionary algorithms (EAs) can be excellent for searching very large spaces, at least when there is some reason to suppose that there are 'building blocks' to be found. A 'building block' is a fragment, in the chosen representation, such that chromosomes which contain it tend to have higher £tness than those which don't. EAs bring building blocks together by chance recombination, and building blocks which are not present in the population at all may still be generated by mutation.

However, the use of EAs are often justi£ed simply by results. If you knew what building blocks looked like in advance, you would not need an EA to bring them together. Nor, usually, are there performance guarantees: in epistatic problems it can happen that the best solutions cannot simply be fabricated from good-looking building blocks. Because of this, EAs often have a problem of acceptability – they look like a 'black box' algorithm that you run until it delivers a solution, but you often do not know whether that solution is even close to optimal, and while it is running you have no easy way to forecast properties of the outcome. The delivered solution may also be fragile, in the sense that there is little continuity between problem speci£cation and EA solution: if you change the problem only slightly, the solution found by re-running the EA changes drastically. It is not surprising therefore that in many practical applications, people may prefer to use a simple heuristic that is comprehensible and perhaps also offers worst-case performance guarantees.

This paper represents a step towards a new way of using EAs that may solve some problems of acceptability for real-world use. The basic idea is as follows: instead of using an EA to discover a solution to a speci£c problem, we use an EA to try to fabricate a solution process applicable to many problem instances and built from simple, well-understood heuristics. Such a solution process might consist of using a certain heuristic initially, but after a while the nature of the remainder of the task may be such that a different heuristic becomes more appropriate.

For example, in [21] an early version of this idea was used to tackle large exam timetabling problems by choosing two heuristics and associated parameters, together with a test for when to switch from using the £rst to using the second. This was motivated by the unsurprising observation that different academic institutions have very different constraints. One institution might have some very large exams, limited exam seating and many smaller exams, so that the important task early on is to pack those large exams together as far as possible in order to plenty of space to deal with placing the many smaller exams. Another institution might have no very large exams, but instead the exams can be clustered such that there are very few inter-cluster con-

straints, and exam clusters can therefore be viewed as relatively independent sub-problems, for which you might naturally choose some other heuristic that placed little emphasis on packing large exams.

An obvious objection to the general idea of hyper-heuristic methods is that, if you combine the use of several heuristics when solving a problem, you will probably lose any worst-case performance guarantee that an individual heuristic had. But against this, there is a simple way to judge the ef£cacy of a composite algorithm against the use of any single heuristic – you might be able to seed the initial EA population with a few chromosomes that represented the process of using only a single heuristic from start to £nish. If such chromosomes do not survive, it is because composite algorithms outperformed them.

In what follows, we describe an example of using hyper-heuristic methods to tackle one-dimensional bin-packing problems. A modern learning classi£er system, XCS [22], is used to learn a set of rules which associate characteristics of the current state of a problem with speci£c heuristics. The set of rules is used to solve problems as follows: given the initial problem characteristics P, a heuristic H is chosen and it packs a bin, thus gradually altering the characteristics of the problem that remains to be solved. At each step a rule appropriate to the current problem state P' is chosen, and the process repeats until all items have been packed.

Using any Michigan-style classi£er system means, of course, that we cannot do what we suggested above and inject pure heuristics into the initial population in order to compare them against composite ones. In a Michigan-style system, the whole population represents one composite algorithm. Nevertheless, XCS represents a simple way to try to fabricate a composite algorithm and the interest lies in seeing how well it can work. In particular, if the system is trained using a few problems, does it then generalise by also performing well on lots of unseen problems? If so (and, to spoil the story, the answer given below is 'yes'), then this is a useful step towards the concept of using EAs to generate strong solution processes rather than merely using them to £nd good individual solutions.

The approach is 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 £nding 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 over 78% of them, and in the rest it produced a solution very close to optimal.

2 ONE-D BIN-PACKING

In the one-dimensional Bin Packing problem (BPP1), there is an unlimited supply of bins, each with capacity c (a positive number). A set of n items is to be packed into the bins, the size of item i is $s_i > 0$, and items must not over-£ll any bin:

$$\sum_{i \in \text{bin}(k)} s_i \leq c$$

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:

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

and for any algorithm that does not start new bins unnecessarily, $M \leq$ bins used $< 2M$ (because if it used $2M$ or more bins there would be two bins whose combined contents were no more than c, and they could be combined into one).

Many results are known about speci£c algorithms. For example, a commonly-used algorithm is First-Fit-Decreasing (FFD): items are taken in order of size, largest £rst, and put in the £rst bin where they will £t (a new bin is opened if necessary, and effectively all bins stay open). It is known [15] that this uses no more than $11M/9+4$ bins. A good survey of such results can be found in [6]. A good introduction to bin-packing algorithms can be found in [18], 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 £nding a combination of 1-3 items that provably does better than anything else (not just any combination of 1-3 items) at £lling a bin, and if so packing them. This may eventually halt with some items still unpacked; the remainder are packed using a 'largest £rst, best £t' algorithm.

Various authors have applied EAs to bin-packing, notably Falkenauer's grouping GA [9, 11, 10]; see also [16, 19] for different approaches. For example, Reeves [19] used a GA to £nd the order in which to feed items to a sequential heuristic such as First-Fit, with reasonable success on a subset of the problems we use in this paper. Falkenauer also produced two classes of benchmark problems. In one of these, the so called *triplet problems*, every bin contains three items; they were generated by £rst constructing a solution which £lled every bin exactly, and then randomly shrinking items a little so that the total shrinkage was less than the bin capacity (thus the same number of bins is necessary).

As ever, speci£c knowledge about problems can help greatly. Suppose you know in advance that each bin contains exactly three items. Take items in order, largest £rst, and for each item search for two others that come very close to £lling the bin. A backtracking algorithm that considers such '£ller pairs', taking pairs in which the two members at most nearly equal in size £rst and permitting only

limited backtracking, solves many of the Falkenauer triplet problems very quickly. See [13] for some questions about whether these problems are hard or not.

The reader may wonder if the simple strategy of searching for a combination of items which come as close as possible to £lling a bin, thereby reducing the problem to a simpler one in which there seems to be more available slack, is a good one. But consider a problem in which bins have capacity 20 and there are six items: 12, 11, 11, 7, 7, 6. One bin can be completely £lled $(7+7+6)$ but then three more bins are needed since the three largest items are each larger than half a bin. If bins are under-£lled, then a three-bin solution is possible, for example $12+7$, $11+7$, $11+6$. We hope this will help to convince the reader that even one-dimensional bin-packing problems have their interest. And they are worth studying because bin-packing is a constituent task of many other optimisation problems; exam timetabling is just one such example.

3 ABOUT XCS

Learning classi£er systems of the Michigan type evolve a set of condition-action rules, by measuring the performance of individual rules and then periodically using crossover and mutation to breed new rules from old. An early account can be found in [14], a more modern account and recent work is in [17].

In early learning classi£er systems, rules occasionally did an action that earned external reward, and this contributed to the rule's £tness and to the £tness of those that enabled it to £re. Earned rewards were spread by the so-called 'bucket brigade algorithm' (effectively a trickle-down economy) or 'pro£t-sharing plan' (essentially a communal reward-sharing) or other such algorithm. However, in those early systems, a rule's £tness was a measure of the reward it might earn (when considering what rule to £re) and also a measure of the reward it had earned (when selecting rules for breeding). This caused various problems, notably that rules which £red very rarely but were crucial when they did would tend to be squeezed out of the population by the evolutionary competition long before they could demonstrate their true value. XCS [22] largely £xed this by instead valuing a rule for the accuracy rather than the size of its prediction of reward.

For this reason – because, in our application, there might be heuristics which were rarely used but crucial – we chose to use XCS rather than, say, Goldberg's SCS.

4 BIN-PACKING BENCHMARK PROBLEMS

We used problems from two sources. The £rst collection is available from Beasley's OR-Library [1], which contains problems of two kinds that were generated and largely studied by Falkenauer [10]. The £rst 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 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 but are not random; the problem generation process was described earlier.

The second class of problems we study in this paper comes from the Operational Research Library [2] at the *Technische Universität Darmstadt*. We used their 'bpp1-1' set and their very hard 'bpp1-3' set 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...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 [18] also used a set with sizes drawn from 50...100, but these are far too easy.) Of these 720 problems, the optimal solution is known in 704 cases and in the other sixteen, the optimal solution is known to lie in some interval of size 2 or 3. 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 $20,000 \ldots 35,000$. The optimal solution is known in only three cases, in the other seven the optimal solution lies in an interval of size 2 or 3. These results were obtained with an exact procedure called BISON [20] that employs a combination of tabu search and modi£ed branch-and-bound.

In all, therefore, we use 890 benchmark problems.

5 COMBINING HEURISTICS WITH XCS

The £rst subsection describes the heuristics we decided to use, and why. The next subsection describes the representation used within XCS. Then we describe how XCS is used to discover a good set of rules.

5.1 The set of heuristics

We £rst evaluated a variety of heuristics to see how they performed on our benchmark collection. Of the fourteen that we tried, some were taken directly form the literature,

others were variants created by us. Some of these algorithms were always dominated by others; among those that sometimes obtained the best of the fourteen results on a problem, some were always £rst equal rather than being uniquely the best of the set. We do not have space here to describe the full set, but we chose to use four whose performance seemed collectively to be representative of the best. These were:

- FFD, 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), see [7]. This puts items into a bin, taking items largest-£rst, until that bin is at least one third full. It then tries to £nd one, or two, or three items that completely £ll the bin. If there is no such combination it tries again, but looking instead for a combination that £lls the bin to within 1 of its capacity. If that fails, it tires to £nd such a combination that £lls 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 modi£ed form of DJD considering combinations of up to £ve items rather than three items. In the Falkenauer problems, DJT performs exactly like DJD, as we would expect; in the bpp1-1 problems it is a little better than DJD.

In addition we also used these algorithms each coupled with a '£ller' process that tried to £nd 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 £lled by using more than three items once the bin was one-third full. Thus, in all we used eight heuristics. The action of the £ller process is described later.

5.2 Representing problem state for XCS

As explained above, the idea is to £nd a good set of rules each of which associates a heuristic with some description of the current state of the problem. To execute the rules, the initial state is used to select a heuristic and that heuristic is used to pack a bin. The rules are then consulted again to £nd a heuristic appropriate to the altered problem state, and the process repeats until all items have been packed.

The problem state is reduced to the following simple description. The number of items remaining to be packed are examined, and the percentage R of items in each of four ranges is calculated. These ranges are shown in Table 1. These are, in a sense, natural choices since at most one

Table 1: Item size ranges

Huge:	items over $1/2$ of bin capacity
Large:	items from $1/3$ up to $1/2$ of bin capacity
Medium:	items from $1/4$ up to $1/3$ of bin capacity
Small:	items up to $1/4$ of bin capacity

huge item will £t in a bin, at most two large items will £t a bin, and so on. The percentage of items that lie within any one of these ranges is encoded using two bits as shown in Table 2. Thus, there are two bits for each of the four

Table 2: Representing the proportion of items in a given range

Bits	Proportion of items
0 0	0 – 10%
0 1	10 – 20%
1 0	20 – 50%
1 1	50 –100%

ranges. Finally, it seemed important to also represent how far the process had got in packing items. For example, if there are very few items left to pack, there will probably be no huge items left. Thus, three bits are used to encode the percentage of the original number of items that still remain to be packed; Table 3 gives the details.

Table 3: Percentage of Items Left

Bits	% left to pack
0 0 0	0 – 12.5
0 0 1	12.5 – 25
0 1 0	25 – 37.5
0 1 1	37.5 – 50
1 0 0	50 – 62.5
1 0 1	62.5 – 75
1 1 0	75 – 87.5
1 1 1	87.5 – 100

The action is an integer indicating the decision of which strategy to use at the current environmental condition, as shown in Table 4. As mentioned earlier, the second four actions use a £ller process too, which tries to £ll any open bins as much as possible. If the £lling action successfully inserts at least one item, the £lling step £nishes. If no insertion was possible, then the associated heuristic (for example, FFD in 'Filler+FFD') is used. This guarantees a

change in the problem state. It is important to remember that the trained XCS chooses deterministically, so that it is important for the problem state (if not the state description) to change each time, to prevent endless looping.

Table 4: The action representation

Action	Meaning, Use
000	FFD
001	NFD
010	DJD
011	DJT
100	Filler + FFD
101	Filler + NFD
110	Filler + DJD
111	Filler + DJT

The alert reader might wonder whether the above problem state description in some way made heuristic selection an easy task. However, when we evaluated each of our 14 original heuristics we found many cases where two problems had the same initial state description but different algorithms were the winners of the 14-way contest. For each of the 14 algorithms we tried using a perceptron to see whether it was possible to classify problems into those on which a given algorithm was a winner and those on which it was not a winner. In every case, it was not possible, and therefore the learning task faced by XCS was not a trivial one.

6 THE EXPERIMENTS

We used Martin Butz' version of XCS [3, 4, 5] available free over the web from the IlliGAL site.

We used a single step environment, in which a reward is available at every step, and we de£ned a step as packing one bin (FFD was modi£ed to pack no more than one bin before returning). The reward earned is proportional to how well £lled that packed bin is. For example if the bin is packed to 94% of capacity, then the reward earned is 0.94. (Following the suggestion of Falkenauer and Delchambre [8], an alternative worth trying in future would be to use the square of this instead). Remember that 'packing' here means continuing to the point where the heuristic would switch bins, rather than optimally packing. Full reward is paid for packing the £nal bin. Otherwise, an algorithm which, say, placed the £nal item of size 1 in a £nal bin in order to complete the packing would earn only 0.01. The £ller is rewarded slightly differently; it is rewarded in proportion to how much it reduces the empty space in the open bins.

The XCS parameters used were exactly as used in [22], with a 50/50 explore/exploit ratio.

For training, we divided each set of bin-packing problems 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. We also combined all problems into one large set of 890 problems and divided that into a training and a test set in the same way. In the results below, we only report on what happened with this combined collection, in which the training set has 667 problems and the test set has 223 problems. Other results are omitted for space reasons; the combined set provides a good test of whether the system can learn from a very varied collection of problems.

The experiments proceeded as follows. We set a limit of L explore/exploit cycles for XCS, where the values we tried were $L = 100, 500, 1000, 5000, 10000, 25000$. During the learning phase, XCS £rst randomly chooses a problem to work on from the training set. One step (whether explore or exploit) corresponds to £lling one bin. In an explore step the action is chosen randomly, in an exploit step it is chosen according to the maximum prediction appropriate to the current problem state description. This is repeated until all the items in the current problem have been packed. A new random problem is then chosen. Clearly, a large problem such as one of the `u1000_M` will consume a great many cycles. We recorded the best result obtained on each problem during this training phase. Remember, however, that training continues, so the rule set may change after such a best result was found. In particular, the £nal rule set at the end of all training might not be able to reproduce the best result on every problem. Nevertheless, it is reasonable to record the best result found during (rather than at the end of) training on each problem, because these are still reproducible results, by re-running the training with the same seed, and easily so.

At the end of training, the £nal rule set is used on every problem in the training set to assess how well this rule set works. It is also applied to every problem in the test set.

7 RESULTS

For the problems we used, details of optimal results are available from [2] and from [1], see Section 4. In the sixteen problems where only a range is known within which the optimal number must lie, we use the upper bound.

The results were as follows:

- during training, XCS found the optimal result for 78.1% of all problems, and for all the others the best result was only one or two bins worse than optimal.

This is encouraging, because for some heuristic algorithms the performance on certain problems can be considerably worse than optimal.

- after £nding a £nal rule set, this was tested. On the training set it found the optimal result on 77.7% of problems. On the test problems, not used during training, it found the optimal result for 74.6% of problems (166 of 223) and again, results were close to optimal on all the rest.

Are these results good? The classi£er system was able to achieve the optimal result in 78.1% of all the benchmark problems, whereas the best single performer of the heuristics considered (namely, our own DJT, introduced in this paper for the £rst time) achieved only 73%. Even though these two results might seem close, it is worth noting that DJT solved none of the very hard bpp1-3 problems while the XCS-generated rule set solved seven out of the ten. It is also noteworthy that, when XCS was trained only on a training set composed of seven of the ten hard bpp1-3 problems, it solved six of those seven, and also one of the three unseen problems. In both cases no other heuristic used alone was able to solve any of these problems.

The worst heuristic is NFD; alone, it was never a winner among the original 14 heuristics we considered. We did include it in the set of heuristics that the classi£er system could invoke, and interestingly it was indeed sometimes invoked as part of a sequence that led to an optimal result, although this happened rarely.

8 CONCLUSIONS AND FUTURE WORK

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

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

- XCS was 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.
- The system always performed better than the worst of the algorithms involved, and in fact produced results that were either optimal (in the large majority of cases) or else were close to optimal.
- The system is able to generalise well. Results of the exploit steps during training are very close to results using a trained classi£er on new test cases. This means that particular details learned (a structure of some kind) during the adaptive phase (when the classi£er rules are being modi£ed according to experience, etc.) can be reproduced with completely new data (unseen problems taken from the test sets). For example, for one of Falkenauer's problems DJD (and our DJT) produced a new best, and optimal, result (this had already been reported in [7] where DJD was described). Even if this problem is excluded from the training set, the learned rule set can still solve it optimally.

In the work reported here we used a single-step environment (reward available after each step). It might be thought that a multi-step environment, with reward proportional to solution quality paid only at the end of a problem or at least after a number of steps were performed. However, learning is likely to be much slower, and we do not even know the number of steps needed to reach a solution in advance. In some problems, such as the `u1000_M`, we have 1000 items to pack and the number of steps to reach a solution and earn any reward could be very large.

We recognise that the reward mechanism perhaps overvalues the £lling of bins, and intend to investigate alternative reward schemes.

Other possible ways to use the multi-step environment could be to allow a chosen rule to continue to perform its action until one of the following happens:

- the problem state has changed so that the rule which chose the action is no longer applicable; or,
- a certain sizeable percentage of items have been placed, eg 20%. This would limit the chain of actions to be at most 5 steps long.

Perhaps also including some extra information about the status of the open bins might be useful. For example, if many open bins contained very little free space and there were many small items still to pack, it might be useful to be able to invoke a heuristic which tried to £ll and £nally close those bins.

Although we have focussed on bin-packing problems in this paper, similar hyper-heuristic ideas could be applied to many other kinds of problem, in which heuristics can be used step by step to transform the problem state from an initial to a £nal one. This raises interesting research questions about how sensitive the approach might be to the choice of heuristics and to the problem state description used.

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] Martin V. Butz. An Implementation of the XCS classi£er system in C. Technical Report 99021, The Illinois Genetic Algorithms Laboratory, 1999.

[4] Martin V. Butz. XCSJava 1.0: An Implementation of the XCS classi£er system in Java. Technical Report 2000027, Illinois Genetic Algorithms Laboratory, 2000.

[5] Martin V. Butz and Stewart W. Wilson. An Algorithmic Description of XCS. Technical Report 2000017, Illinois Genetic Algorithms Laboratory, 2000.

[6] 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.

[7] Philipp A. Djang and Paul R. Finch. Solving One Dimensional Bin Packing Problems. *Journal of Heuristics*, 1998.

[8] E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. In *Proc. of the IEEE 1992 International Conference on Robotics and Automation*, pages 1186–1192, 1992.

[9] Emanuel Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary Computation*, 2(2):123–144, 1994.

[10] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996. http://citeseer.nj.nec.com/falkenauer96hybrid.html.

[11] 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.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[13] I. P. Gent. Heuristic Solution of Open Bin Packing Problems. *Journal of Heuristics*, 3(4):299–304, 1998.

[14] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA., 1989.

[15] D.S. Johnson. *Near-optimal bin-packing algorithms*. PhD thesis, MIT Department of Mathematics, Cambridge, Mass., 1973.

[16] 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, *Arti£cial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Ales, France, 1995*, 1995.

[17] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classi£er Systems: From Foundations to Applications*, volume 1813 of *Lecture Notes in Arti£cial Intelligence*. Springer-Verlag, Berlin, 2000.

[18] Silvano Martello and Paolo Toth. *Knapsack Problems. Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[19] C. Reeves. Hybrid genetic algorithms for bin-acking and related problems. *Annals of Operations Research*, (63):371–396, 1996.

[20] Armin Scholl and Robert Klein. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research*, 1997.

[21] Hugo Terashima-Marín, Peter Ross, and Manuel Valenzuela-Rendón. Evolution of constraint satisfaction strategies in examination timetabling. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 635–642. Morgan Kaufmann, 1999. early hyper-heuristic.

[22] Stewart W. Wilson. Classi£er Systems Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.