Policy Matrix Evolution for Generation of Heuristics

Ender Özcan
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, U.K.
exo@cs.nott.ac.uk

Andrew J. Parkes[*]
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, U.K.
ajp@cs.nott.ac.uk

ABSTRACT

Online bin-packing is a well-known problem in which immediate decisions must be made about the placement of items with various sizes into fixed capacity bins. The associated decisions can be based on an index policy in which each decision option is independently given a value and the highest value choice is selected. In this paper, we represent such heuristics for online bin packing as a simple matrix of scores. We then use a genetic algorithm to search for matrices giving good performance. This might be regarded as parameter tuning of the packing heuristic but in which a fine-grained representation is used and so the number of parameters is much larger than in standard parameter tuning. The evolved matrices perform better than the standard heuristics. They also reveal interesting structures and so have impact on questions of how heuristic score functions should be represented and what structure they might be expected to exhibit.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: ARTIFICIAL INTELLIGENCE—*Problem Solving, Control Methods, and Search*

General Terms

Algorithms

Keywords

Heuristics, hyper-heuristics, metaheuristics, index policies, online problems

1. INTRODUCTION

In many situations, decisions must be made despite lack of knowledge of the future or lack of computational resources to precisely compute the effects of the decisions. In such cases, it is usual to have some kind of heuristic to make decisions. Usually, heuristics are produced by an expert in

---

[*]Contact Author.

a domain carefully designing some decision procedure. Often, even an expert requires a great deal of trial and error - though the errors are rarely reported, and so a misleading impression given suggesting that creation of heuristics is not a time-consuming process. Of course, such difficulties are well-known, and so there have been various attempts to automate the production of heuristics (e.g. for some recent work see [20, 8, 5]).

In this paper, we give an approach for the automatic creation of heuristics that might be viewed as a form of parameter tuning [24] but applied with a much larger number of parameters than usual. The large number of parameters arise from a 'brute force' representation of the heuristic as a matrix covering the various potential decisions, that is, it defines a policy in terms of the 'features' available at each decision point. This is done in the style of an 'index policy' [14] in that each potential outcome is given a score separately of other outcomes and the largest score is selected.

In particular, we study the well-known online bin-packing problem [9, 11], creating a policy that is based on using a (large) matrix of 'heuristic scores'. Packing problem instances are specified in terms of the bin size and the range of item sizes. For specific instances, good policies are found using a Genetic Algorithm (GA) to search the space of matrices, with the matrix-based policies being evaluated directly by packing a (large) number of items.

The first result is that the GA finds matrices for the specific packing problems that perform significantly better than the standard general-purpose heuristics such as first and best fit [19, 10]. We then endeavour to understand and explain the improved performance by looking at the structure of the resulting good policy matrices. We find that they have a surprisingly 'spiky' and complex structure that is quite unlike the simple structure associated with the standard heuristics. The structure also does not seem to be a good match to a simple arithmetic function of the score as a function of the features. This could well help understand cases when previous approaches using Genetic Programming [4, 6] were unable to obtain similar improvements. Hence, this work raises general questions about the need to pick appropriate representations of index policies. The advantage of a matrix-based approach is that it can be agnostic about the form of the index values, and so not exclude potential policies by making an inappropriate choice of representation. Generally this suggests that when using evolutionary (or other methods) to discover heuristics that it is important not to presume biased representations before looking at unbiased ones in detail to discover the structures that good heuristics

ought to have. Of course, a potential disadvantage is that a matrix based policy might lead to large number of parameters, and so make the search task computationally challenging for a general-purpose GA. Specialized search methods might of course help with this challenge. However, even if very large instances were to remain impractical, our motivation is that heuristics could be generated on small to medium instances, and their structure inspected and used as the basis for creation of heuristics for larger instances.

We are of course aware that standard methods for policy creation in stochastic processes (Markov chains, reinforcement learning, etc) are also likely to be able to generate good policies. However, our driving motivation is to form the basis for (evolutionary-)search methods to aid in the generation of heuristics and heuristic policies for complex situations (and out of the reach of analytical methods). Such complex situations might include combinatorial optimisation problems using constructive heuristics, or queuing networks.

The structure of the paper is as follows: Section 2 gives a brief review and pointers into the literature of existing work on firstly bin-packing and also computer-based methods to help design heuristics. Section 3 gives basic definitions of the bin-packing problem, the instances that we use, and the existing standard heuristics. Section 4 specifies the matrix-based framework we use in order to define the packing policies. Section 5 describes the GA search method we use to find good policies. Section 6 describes in depth a simple illustrative example in which asymptotically optimal policies are easy to find and understand. Section 7 gives the main results on examples that are large enough to allow policy improvements, but still small enough that the structure of the resulting policies can be (partially) understood. Section 8 summarises our results and their implications, and then discusses future plans.

2. RELATED WORK

We now give a brief review and pointers into the literature of existing work, firstly on bin-packing and then on computer-based methods to help design heuristics.

2.1 Bin-packing

One dimensional bin packing is a combinatorial optimisation grouping problem, proven to be NP-hard [13]. This problem involves packing a number of pieces with given sizes into a minimal number of bins, where every bin has the same fixed capacity. In other words, a set of integers must be partitioned into subsets (groups) so that the sum of the integers within a subset does not exceed the capacity. In this generic offline problem, the solution approaches have complete information about the number of pieces and their sizes. Heuristics are commonly used for bin packing [10], whenever exact algorithms fail to produce a result in reasonable time for a given problem. There are also other studies addressing the representation issues for bin packing in metaheuristics. For example, [12] used group encoding for candidate solution representation in the framework of genetic algorithm and combined the grouping genetic algorithm with local search based on the Martello and Toth's branch and bound reduction algorithm [16]. Ülker et al. [25] presented a linear linkage encoding to overcome the redundancy in the group encoding representation within another grouping genetic al-

gorithm framework. More on bin packing can be found in [9, 11].

Online bin packing problem is a well known variant in which the pieces arrive sequentially and at each step, a packing decision has to be made before the next item size is revealed [23]. The decision is made under incomplete information about the number of pieces and their sizes, and results in putting the current item into an already open bin or opening a new bin.

2.2 Existing methods to discover heuristics

Hyper-heuristics provide search and optimisation frameworks which allow the exploration of heuristics space for solving complex problems [2, 20, 17, 8]. There are two main classes of hyper-heuristics; methodologies that *select* or *generate* low level heuristics [3]. Genetic Programming (GP) is an Evolutionary Algorithm that searches the space of computer programs. It has been applied to many different challenging problems [18]. GP has been used as a hyper-heuristic for the automated design of heuristics [5].

In [21], a learning classifier system of the Michigan type XCS was trained to generate a hyper-heuristic for solving unseen bin packing problem instances. The system learns how to choose a low level heuristic to iteratively construct a solution starting from an initial state towards a final state, in which no item is left for packing. The authors reported that XCS performed well across a large collection of data. In this study, the number of training instances used was much larger than the test cases.

Particularly relevant is the work in [4] that used GP to evolve heuristics to decide the choice of bin for packing a given item. The results showed that the first fit heuristic could be generated by the genetic programming approach. The authors reported that the code-bloat [1] existed in their genetic programming. Moreover, they confirmed the redundancy in the representation, as GP generated four trees with depth of 2 showing similar performances to the first fit heuristic. Subsequent work in [7] obtained new heuristics after a training phase using genetic programming and tested these online bin packing heuristics on randomly generated problem instances. The authors investigated the behavior of the computer generated heuristics as the problem size increases against different sizes of training problem instances. As expected, they have observed that the performance of generated heuristics improves as the size of training problems and test problems increase. The new heuristics produced a competitive performance to the best fit heuristic. The same authors [6] further studied the trade off between performance and level of generality of a heuristic generated by genetic programming for online bin packing. The authors have illustrated that indeed there is a trade-off and the representative problem instances used during the training phase is crucial.

3. ONLINE BIN PACKING PROBLEM

This section gives the definitions, notation and standard heuristics for online bin-packing that form the context for the rest of the paper.

Basic definitions.

All bins will be taken to have the same integer capacity $C > 1$ that is a hard limit on the total of the sizes of the items that it can contain. Item sizes are also positive integers

within the range $[1, C-1]$. Items arrive one at a time, and must be assigned irrevocably to a bin before the next item is seen.

A single empty new bin is always guaranteed to be available. If an item is packed into the empty bin then it is said to have been opened and a new empty bin is created. A bin will is referred to as closed if the remaining space is too small to handle any potential item. Otherwise, non-empty bins will be said to be open. (In some circumstances, an item might open the empty bin and simultaneously close it because there is not enough space for any other item.)

Standard heuristics.

Well known heuristics are First Fit (FF), Best Fit (BF) and Worst Fit (WF) [15, 19, 10]. Specifically, we take

FF First fit packs the item in the earliest available bin that has sufficient space to take it. The motivation is that, in the long run, earlier bins will tend to be packed well and then closed.

BF Best fit packs the item in the fullest available bin that will take it. Ties are broken using FF (take the earliest fullest bin). The motivation is to encourage each bin to be packed perfectly or as near as possible to perfect.

WF Worst fit packs an item into the emptiest available non-empty bin. The motivation is to leave space for other items (not surprisingly it is not a very good heuristic). Ties are again broken using FF.

Uniform Bin Packing Instances.

We will use instances with N items in total, bin capacity C, and integer item sizes selected uniformly and independently at random from the range from the interval $[s_{min}, s_{max}]$. For compactness we will refer to these as

$$\text{UBP}(C, s_{min}, s_{max}, N)$$

We will assume $s_{min} > 0$ and $s_{max} < C$ as items of zero size can be ignored, and items of size C or larger do not need a policy to handle them.

Some of the data sets of [22] and [12] as included in the OR library are generated similarly, however are smaller in terms of number of items as they are designed for testing offline algorithms. Hence, we simply generated our own random instances.

Performance Measures.

Performance of packing policy will be measured using one or more of the following:

Bins-Used, B: Simply the number of bins that are used. Although having the convenience of being an integer it will increase with the number of items N, and not be useful as N grows large. So assuming that bin t has fullness f_t, $t \in \{1, \ldots, B\}$, it is more convenient to report:

Average-Fullness, F_{af}: The overage, over the bins that have been used of the space occupied to capacity. $F_{af} = 1/B \sum_t f_t$

Average-Generic-Fullness, F_{gf}: This value gives some insight into the variation of resulting fullness between bins. $F_{gf} = 1/B \sum_t f_t^2$

Average-Perfection, F_{ap}: This value indicates how successful the packing heuristic is in filling bins perfectly. $F_{ap} = 1/B \sum_{t, f_t=1} f_t$

In this paper we will search for policies that maximise the average fullness, though we will also report the generic fullness and average perfection as they can give insight into how the policy is behaving. (Maximising the generic fullness would lead to packing that are more fairly distributed between bins.)

4. VALUE MATRIX FRAMEWORK

We will define the packing policy using a matrix of heuristic values:

W_{ci} is the integer value (score) associated with assigning an item of size i to a bin of remaining capacity c.

The algorithm for packing an incoming item of size i is then simply:

1. For each open bin (including the always-available new empty bin), find its remaining capacity c. If it satisfies $c \geq i$, so that the item might be placed in the bin, then give the bin a value of W_{ci}.

2. The item is placed into the bin with the largest value. We take ties to be broken using the FF heuristic. That is, the earliest highest-scoring bin is selected.

We will take the values W_{ci} from a range $[w_{min}, w_{max}]$ which is chosen according to the instance. Generally, the aim will be to keep the range as small as possible while retaining sufficient expressive power.

It is straightforward to generate matrices that correspond to the standard heuristics. For first-fit all values are equal, and then the tie-breaking rule always applies. For best fit values increase as the diagonal is approached, and for worst fit values decrease as the diagonal is approached. This will be illustrated in the small example in section 6; see equations (3)-(5).

It is important to observe that the matrix representation is general and fair. A wide range of policies are expressible. In this paper, for simplicity we use FF to break ties, but in general other tie breaking rules could be considered. The system is fair in the sense that all policies correspond to a matrix of the same size, and so there is no bias towards smoother or 'nicely-structured' heuristics. Such 'learning without preconceptions' can allow it to reach heuristics and policies that might not otherwise be considered.

Notice that each column of the matrix essentially defines a policy for how the corresponding item size is handled. The GA search procedure allows columns to be independent of each other; and so a policy for size i is permitted to be quite different from that for size $i + 1$. We will see later that this 'column independence' is important.

Generally we will be working with instances in which item sizes are within a restricted range. In this case some entries within the value matrix W_{ci} will be irrelevant because the pair (c, i) can never occur, or will be irrelevant. For example, $i > c$ is irrelevant as the item is too big to fit. We also only need to consider values of i corresponding to item sizes that can occur. In practice this means that the search for a good matrix only needs to consider a subset of the values. We represent this in practice using a boolean "mask matrix"

M_{ci} that is true if and only if W_{ci} might be used within the instances considered. In the explicit examples in later sections we will use a dot '.' for the irrelevant unused entries in the matrices. From now on, reference to a matrix will be taken to mean that only the used, un-masked, entries are considered.

4.1 Normalisation of the Value Matrix

The decisions made during packing depend only on relative values given to bins. For example, doubling all values of W_{ci} will always give exactly the same packing.

Hence, many different matrices are equivalent to each other. The search process we use does not use any criteria to select between different elements of such equivalence classes. This has the practical impact that the raw matrices it produces are harder to interpret and compare. Hence, we defined and used a rewrite procedure to convert matrices to a standard form selects a unique representative for each equivalence class and having a structure that is easier to interpret.

Recalling that the columns of the matrix are essentially independent, then we just need rewrite rules for a column. The primary rules we use, and applied to each value of i separately, are as follows:

1. Uniformly add to all scores to guarantee the score of the empty bin $W_{Ci} > 1$

2. The empty bin, of capacity C, is always available, so if some other capacity $c < C$ has a lower score then it will never be selected. Hence, given the first rule, we can now re-assign the scores of such unusable capacities to a fixed value: $W_{ci} = 1$.

3. The empty bin is always considered last, so if it is in a tie-break (using FF) it will be last choice. Hence a bin with a higher score can be reduced to the same score as the empty bin, as long as this does not change its order relative to other values of c.

4. Remove all gaps in the set of values in the column, and at the same time reduce them as much as possible without changing their relative ordering.

5. EVOLVING POLICY MATRICES

We have seen that the packing policy is determined by a simple matrix, and an associated mask matrix saying which entries are active in the sense that they might be used on the set of instances under consideration. Hence, given an instance of packing (as determined by the bin capacity and the stream of items), evaluation is simply done by running the policy on the instance and measuring performance using one (or more) of the measures in Section 3.

We optimise the matrix using an offline learning Genetic Algorithm (GA) with the individuals (candidate solutions) representing the active members of the policy matrix. Hence, each gene carries an allele value in $[w_{min}, w_{max}]$. The population of individuals goes though the usual cyclic evolutionary process of selection, recombination, mutation and evaluation. The individuals for crossover is selected based on tournament selection with a tour-size of 2. Uniform crossover which exchanges each gene from parents with a probability of 0.5 is used. The crossover probability is set to 1.0. The traditional mutation goes over each gene successively and randomly assigns a new value from $[w_{min}, w_{max}]$ with a probability of $1.0/individual\_length$, where the $individual\_length$ is the number of the active members of the policy matrix. As illustrated in Figure 1, the control function generator and fitness evaluator are physically separated in the implementation for generality, flexibility and extendibility purposes. The GA and the fitness evaluator communicate through text files: The GA saves an individual into a file in a matrix form and invokes the online bin packing program. The packing algorithm uses the matrix as a policy as described in Section 4 and evaluates its quality using a set of 50 different training instances of UBP(C,s_{min},s_{max},500). The total number of bins used while solving each training case is accumulated and then saved as the fitness of the individual into another file for GA to read from. The initial population is randomly generated and the training process continues until a maximum number of iterations is exceeded.
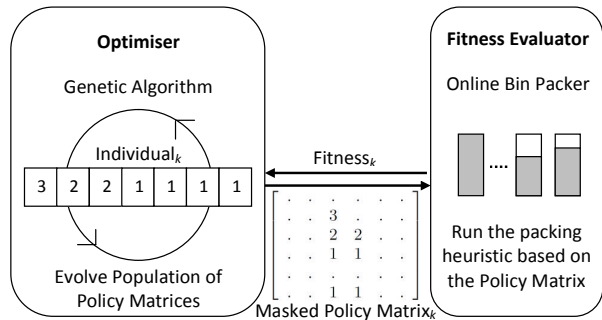

Figure 1: Heuristic generation using a GA for the online bin packing problem.

A policy matrix represents a heuristic. Hence, here the GA is used as a hyper-heuristic searching for a good heuristic for online bin packing. A hyper-heuristic operates at a domain-independent level and does not access problem specific information (e.g. see [20]), thus, the framework we use, as shown in Figure 1, follows the same structure.

The resultant best (B-PM) and worst (W-PM) performing matrices (constructive heuristics) from 50 runs are then tested on randomly generated 100 different instances of UBP(C,s_{min},s_{max},10^5). The goal is to observe how the learned policies perform when compared to the human designed heuristics for online packing on unseen problem instances drawn from the same distribution. The experiments are conducted on 2.1GHz dual core computers with 2GB memory.

In this paper, we fix $w_{min} = 1$, while for each instance, w_{max} is set to the largest number of active entries among the columns of the masked policy matrix. This means that each active entry in every column can be given a different value and so any preference ordering can be expressed for any (used) item size.

In Section 7, we will report the results of applying the framework to three problems:
UBP($6,2,3,10^5$), UBP($20,5,10,10^5$) and UBP($40,10,20,10^5$). The GA uses a population size of 4, 12 and 24 for each instance, respectively, and so each iteration (except the first) requires 2, 10 and 22 fitness evaluations (FEs). The maximum number of iterations is 200, and so the maximum number of FEs per training run are 402, 2002 and 4402,

respectively. Each FE during training with 500 items typically takes less than 10ms. (The testing phase with 10^5 items takes less than 1 second.) Wall-clock run-times are generally around a minute per training run, however, we do not report the exact time as they are relatively small and we have not tuned the GA parameters. Also, the implementation started a new process for each FE and so we suspect significant time is spent on file read/write operations. This could be improved by combining the GA and online bin-packing codes.

6. AN ILLUSTRATIVE CASE: UBP(6,2,3)

In order to illustrate the ideas we will first look in detail at the simple example UBP(6,2,3) (we suppress the last entry for number of items when no specific value is intended). That is, bins of capacity 6 and items of size 2 or 3 only.

Firstly, observe that just two perfect packings are available: 2+2+2 or 3+3. Also, there is only one packing, 2+3, that results in wasted unusable space. Hence, in this case it is straightforward to invent a good heuristic for a policy: Never mix the two item sizes in the same bin.

Open bins will then fall into two classes;

- "even": these have one or two items of size=2, and are 'waiting' for another size 2 item

- "odd": these have a single size=3 item and waiting for another size 3

Of course, it also then makes sense for incoming items to be placed in the most full bin of the correct parity. This policy in terms of the of the value matrix is simply:

$$W_1 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 2 & \cdot & 3 & \cdot & \cdot & \cdot \\ 3 & \cdot & 1 & 2 & \cdot & \cdot \\ 4 & \cdot & 2 & 1 & \cdot & \cdot \\ 5 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 6 & \cdot & 2 & 2 & \cdot & \cdot \\ \hline c/i & 1 & 2 & 3 & 4 & 5 \end{bmatrix} \quad (1)$$

where temporarily, for clarity, we have added labels to the rows and columns giving the remaining space, c, in the bin, and the item size, i, respectively.

Interpreting such matrices can be done column-wise, as a column defines the policy for a given item size. For example, $W_{43} = 1$ compared to $W_{63} = 2$ means that for items of size 3, a remaining capacity of c=4 is never taken, and instead a new bin is opened. Since c=4 can only arise from having put a size=1 item, then this matches the non-mixing of items. The tie $W_{33} = W_{63}$ is broken using FF, and so is always won by the case c=3, as open bins are always earlier than the empty bin.

In this case, it happens that the following similar matrix will behave essentially identically:

$$W_2 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ \cdot & 1 & 2 & \cdot & \cdot \\ \cdot & 2 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & 2 & \cdot & \cdot \end{bmatrix} \quad (2)$$

The change from W_{22} from 3 to 2 means that for item size=2, a remaining space of 2 is no longer strictly preferred over

one of 4. However, because ties are broken using FF then naturally the cases of 2 will occur first, and so will be taken.

These hand-designed policies will give an essential perfect packing (only being imperfect with the last few items). Since the sizes are kept separate then all the 2s will end up together allowing 2+2+2=6, and similarly the 3s will stay together and give a 3+3=6. Also notice that in this particularly simple case with items being of distinct classes, the relative probability of each item size does not matter (ignoring possible boundary cases). Sizes 2 and 3 should be packed separately independently of the ratio of their frequencies.

As mentioned in section 4, the standard heuristics can also be expressed as matrices. In this example, they are as follows:

$$W_{FF} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \end{bmatrix} \quad (3)$$

$$W_{BF} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 3 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 2 & 2 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \end{bmatrix} \quad (4)$$

$$W_{WF} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 2 & 1 & \cdot & \cdot \\ \cdot & \cdot & 3 & 2 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \end{bmatrix} \quad (5)$$

None of these standard heuristics will keep the item sizes separate. If a single item of size=2 has arrived, and then an item size=3 arrives, the standard heuristics will all place it in the bin with the 2, rather than opening a new bin. Table 1 gives their performances, and they achieve a relatively poor fullness.

Table 1: Comparison of the standard heuristics FF, BF, WF, with a selection of the matrix-based policies found by the GA search for solving UBP(6, 2, 3, 10^5)

Heuristic	F_{af}	F_{gf}	F_{ap}
FF	0.92251	0.85794	0.53511
BF	0.92251	0.85794	0.53511
WF	0.91658	0.84707	0.49952
B-PM	**0.99999**	**0.99998**	**0.99997**
W-PM	0.96273	0.93167	0.77640

On the same instance, we have also run the training system of Section 5 in order to see whether or not better policies can be found automatically. With multiple training runs, Table 1 reports the performance of the best policy matrix, B-PM, and worst policy matrix, W-PM, of the training runs. Even the worst of the evolved matrices outperforms the standard heuristics. The best runs often find the

hand-crafted heuristics above, and achieve essentially perfect packing. The only imperfections arise from horizon effects on the last few items. For example, if the last item is size 3 then it should be packed into a c=4 if possible, but the policy might wastefully open a new bin.

7. COMPUTATIONAL RESULTS

Here we first report on the main results of experiments using of generating value matrices for specific problems using the evolutionary search methods given in section 5, showing they perform better than the general heuristics. We then analyze and interpret the matrices to see why they might be doing so much better.

Table 2: Results for UBP(20, 5, 10, 10^5). Comparing the standard heuristics FF, BF, and WF, with the best, B-PM, and worst, W-PM, of the runs of the matrix evolution according the the performance criteria in section 3.
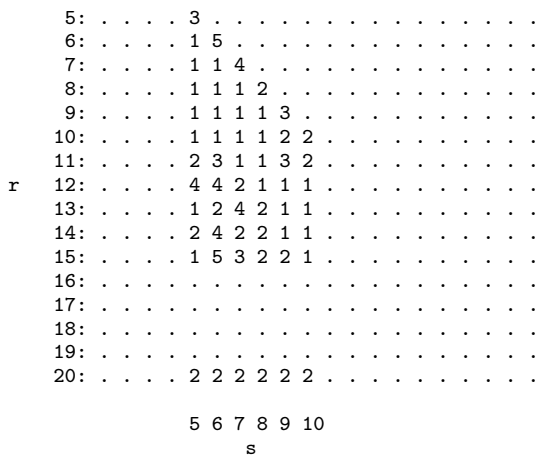
Heuristic	F_{af}	F_{gf}	F_{ap}
FF	0.91536	0.84340	0.31739
BF	0.91549	0.84365	0.31983
WF	0.90545	0.82608	0.27505
B-PM	**0.98433**	**0.96983**	**0.75897**
W-PM	0.97305	0.94879	0.62763

Table 3: Results for UBP(40, 10, 20, 10^5) in the same format as table 2.

Heuristic	F_{af}	F_{gf}	F_{ap}
FF	0.90224	0.82009	0.20765
BF	0.90238	0.82036	0.21191
WF	0.88669	0.79289	0.13565
B-PM	**0.97124**	**0.94506**	**0.57340**
W-PM	0.95867	0.92233	0.57184

The instances we will look at here are UBP(20, 5, 10, 10^5) and UBP(40, 10, 20, 10^5). These are selected because initial experiments showed that they were cases in which the usual heuristics perform particularly poorly, and so most clearly illustrate the potential gains from the overall method. (However, we intend to report a much broader study elsewhere.) The results for these two instances are summarised in tables 2 and 3. In both tables, we give results for the standard heuristics, and then the performances of best matrices found in the best run B-PM and worst runs W-PM of the matrix evolution. Performances are given in terms of average and generic fullness and also the average perfection as described in section 3. The results are given to 5 decimal places as the statistical variance is very small due to the large number, 10^5, of items that are packed in the testing phase.

The first observation is that in all cases even the worst results from the matrix evolution (with statistical significance) outperforms the standard heuristic. (That every one of the 50 training runs outperform the standard heuristics shows the statistical significance of the results.) Furthermore, the policy generated from the best runs of the evolved heuristic substantial outperform the standard ones. Perhaps most striking is the difference in the average perfection; the

```
 5: . . . . 3 . . . . . . . . . . . . .
 6: . . . . 1 5 . . . . . . . . . . . .
 7: . . . . 1 1 4 . . . . . . . . . . .
 8: . . . . 1 1 1 2 . . . . . . . . . .
 9: . . . . 1 1 1 1 3 . . . . . . . . .
10: . . . . 1 1 1 1 2 2 . . . . . . . .
11: . . . . 2 3 1 1 3 2 . . . . . . . .
r  12: . . . . 4 4 2 1 1 1 . . . . . . . .
13: . . . . 1 2 4 2 1 1 . . . . . . . .
14: . . . . 2 4 2 2 1 1 . . . . . . . .
15: . . . . 1 5 3 2 2 1 . . . . . . . .
16: . . . . . . . . . . . . . . . . . .
17: . . . . . . . . . . . . . . . . . .
18: . . . . . . . . . . . . . . . . . .
19: . . . . . . . . . . . . . . . . . .
20: . . . . 2 2 2 2 2 2 . . . . . . . .

        5 6 7 8 9 10
              s
```

Figure 2: One of the best evolved matrices for UBP(20,5,10), labeled by residual capacity r and item size s.

evolved policies are a lot more successful at managing to totally fill bins.

7.1 Structure of Good Policies

Given the success shown above of the evolved matrices it is natural to look at them in detail to try to understand why they are so much better. For the UBP(20,5,10) instance, one of the resulting best matrices is given in Figure 2 where again we have converted to the normalized form and the dots are unused inactive entries. Since reading the matrix directly is not very easy we also give a colour map version in figure 3. The picture is restricted to the active part of the matrix and we also show the colourmaps for the standard heuristics to allow a visual comparison.

For the winning evolved matrix, perhaps the most striking feature is the absence of a striking structure.

The low values for entries near to, but not on, the diagonal are understandable as their use would correspond to choices that leave bins with some unusable space and so ought to be avoided. However, unlike the standard heuristics the matrix seems to have more 'spikes' in the middle of the range of remaining capacity values. Furthermore, the spikes in different columns are not neatly aligned. For example, item sizes 6 and 7 have quite different sets of preferred values for the remaining capacities. Our hypothesis is that the structure of 'non-aligned spikes' manages to precisely exploit the probability distribution of item sizes and so assign them in a carefully interlocking fashion to different remaining capacities.

Figure 7.1 gives a picture for the best evolved matrix for the case of UBP(40,10,20,10^5) and shows that the spiky structure is not an accident of a single case.

7.2 Evolving Hyper-Heuristic Policies

Briefly, we mention that an alternative to using matrix-based heuristics would be to have a row matrix, indexed by items size i and with entries $W_i \in \{FF, BF, WF\}$ used to select which heuristics should be applied for that item size. One might conjecture that a suitable W_i might have a good mix of selections that leads to an improved policy. We have also experimented with this approach, but it failed
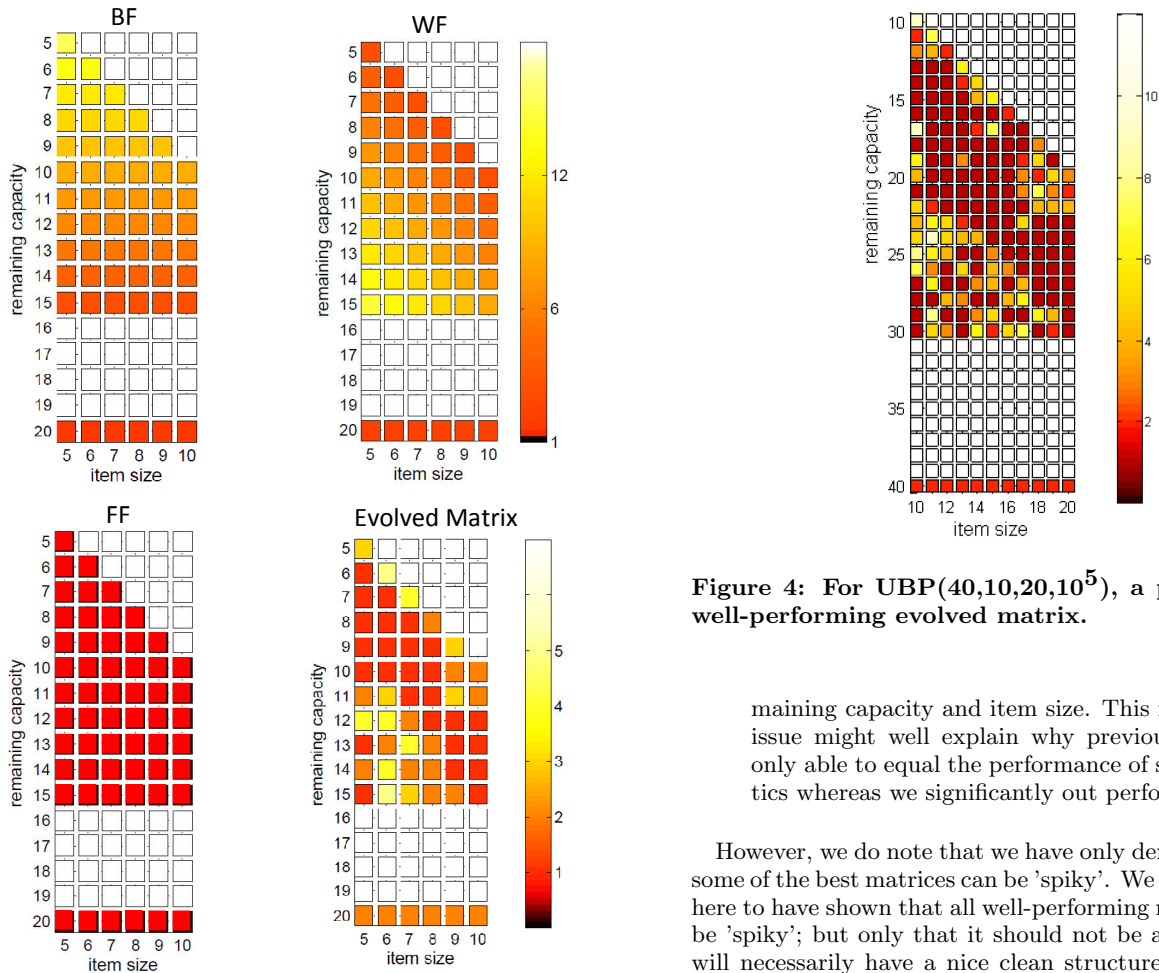
Figure 3: **Pictures of the active areas of matrices for FF, BF, WF and one of the best evolved matrices.**

Figure 4: **For UBP(40,10,20,10^5), a picture of the well-performing evolved matrix.**

to give any significant improvements, and was substantially outperformed by the methods given above. This gives some evidence, at least in this case, for the view that it is better to generate (index) policies rather than rely on trying to make clever selections between existing ones.

8. CONCLUSION

We have found that if we use a policy defined by a simple matrix of score values, then a standard GA approach can produce policies tuned to the instances under consideration and substantially outperforming the generic heuristics such as first and best fit. Inspection of the discovered policies, as in figures 3 and 7.1, shows a "non-aligned spiky" structure. This gives evidence for two main conclusions:

1. Although such matrices could be converted into separate packing rules for each item size, and might well be understandable *post facto*, it also seems that the correlations needed between item sizes would be hard to create by hand. It is unlikely that such heuristics would be designed by a human; giving evidence for the need for their search-based discovery.

2. It does not seem likely that the structures will be easy to represent using a 'nice' arithmetic function of re-

maining capacity and item size. This representational issue might well explain why previous work [4] was only able to equal the performance of standard heuristics whereas we significantly out perform them.

However, we do note that we have only demonstrated that some of the best matrices can be 'spiky'. We are not claiming here to have shown that all well-performing matrices need to be 'spiky'; but only that it should not be assumed policies will necessarily have a nice clean structure. It might well be that 'nicer' matrices do also exist and maybe also have a good performance. Possibly, there is a tradeoff between the 'smoothness' of the matrix and the performance. Investigation of such a 'smoothness' vs. performance tradeoff is currently under investigation. An advantage of the matrix representation is that it allows such issues to be explored.

Notice that in the case of UBP(40,10,20,10^5) the policy matrix contains 147 active entries, and each one can be regarded as a different parameter of a policy. Hence, this approach may be viewed as a form of parameter tuning [24] but with a large number of parameters taking discrete values from a relatively small domain. This makes it more like discrete optimisation on a high dimensional space than the continuous optimisation of a lower-dimensional space. We suspect that this difference is ultimately likely to have effects on the search methods.

Obvious avenues for future research are: To study bigger and more varied instances; improve the search methods used to discover good matrices; and apply the general approach to other problem domains.

Perhaps, most importantly, the mechanisms of landscape analysis (fitness-distance correlation, etc) can be applied to the space of matrices. Since matrices represent heuristics, this opens the intriguing possibility of deepening the understanding of the space of heuristics using well-established techniques from the area of evolutionary search.

9. REFERENCES

[1] Y. Bernstein, X. Li, V. Ciesielski, and A. Song. Multiobjective parsimony enforcement for superior generalisation performance. In G. Greenwood, editor, *Proceedings of the IEEE Congress on Evolutionary Computation (CEC2004)*, pages 83–89. 2004.

[2] E. K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 457–474. Kluwer, 2003.

[3] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research and Management Science*, pages 449–468. Springer US, 2010.

[4] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006)*, volume 4193 of *Lecture Notes in Computer Science*, pages 860–869, Reykjavik, Iceland, September 2006.

[5] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In J. Kacprzyk, L. C. Jain, C. L. Mumford, and L. C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, pages 177–201. Springer Berlin Heidelberg, 2009.

[6] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1559–1565, New York, NY, USA, 2007. ACM.

[7] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. The scalability of evolved on line bin packing heuristics. In D. Srinivasan and L. Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages 2530–2537, Singapore, 25-28 Sept. 2007. IEEE Computational Intelligence Society, IEEE Press.

[8] K. Chakhlevitch and P. Cowling. Hyperheuristics: Recent developments. In C. Cotta, M. Sevaux, and K. Sörensen, editors, *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies in Computational Intelligence*, pages 3–29. Springer Berlin / Heidelberg, 2008.

[9] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[10] E. G. Coffman Jr, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Combinatorial analysis. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1 of *Intelligent Systems Reference Library*, pages 151–207. Kluwer Academic Publishers, 1999.

[11] J. Csirik and G. Woeginger. On-line packing and covering problems. In A. Fiat and G. Woeginger, editors, *Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 147–177. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0029568.

[12] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996. 10.1007/BF00226291.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[14] J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41(2):pp. 148–177, 1979.

[15] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

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

[17] E. Özcan, B. Bilgin, and E. E. Korkmaz. A comprehensive survey of hyperheuristics. *Intelligent Data Analysis*, 12(1):3–23, 2008.

[18] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. (With contributions by J. R. Koza).

[19] W. T. Rhee and M. Talagrand. On line bin packing with items of random size. *Mathematics of Operations Research*, 18(2):pp. 438–445, 1993.

[20] P. Ross. Hyper-heuristics. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 17, pages 529–556. Springer, 2005.

[21] P. Ross, S. Schulenburg, J. G. Marín-Blázquez, and E. Hart. Hyper-heurstics: learning to combine simple heuristics in bin-packing problems. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 942-948, New York NY, 2002. Morgan Kauffmann Publishers.*, 2002.

[22] A. Scholl, R. Klein, and C. Jürgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627 – 645, 1997.

[23] S. S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49:640–671, September 2002.

[24] S. K. Smit and A. E. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 399–406, Piscataway, NJ, USA, 2009. IEEE Press.

[25] O. Ülker, E. Korkmaz, and E. Özcan. A grouping genetic algorithm using linear linkage encoding for bin packing. In G. Rudolph, T. Jansen, S. Lucas, C. Poloni, and N. Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pages 1140–1149. Springer Berlin / Heidelberg, 2008.