A HyFlex Module for the One Dimensional Bin Packing Problem

Matthew Hyde, Gabriela Ochoa, José Antonio Vázquez-Rodríguez, Tim Curtois
Automated Scheduling, Optimisation and Planning (ASAP) Group, School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham. NG8 1BB. UK

1 Problem Formulation

The classical one dimensional bin packing problem consists of a set of pieces, which must be packed into as few bins as possible. Each piece j has a weight w_j, and each bin has capacity c. The objective is to minimise the number of bins used, where each piece is assigned to one bin only, and the weight of the pieces in each bin does not exceed c. To avoid large plateaus in the search space around the best solutions, we employ an alternative fitness function to the number of bins, which is explained in section 5. A mathematical formulation of the bin packing problem is shown in equation 1, taken from [1].

$$\begin{aligned}
&\text{Minimise} && \sum_{i=1}^{n} y_i && \\
&\text{Subject to} && \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, && i \in N = \{1, \ldots, n\}, \\
& && \sum_{i=1}^{n} x_{ij} = 1, && j \in N, \\
& && y_i \in \{0,1\}, && i \in N, \\
& && x_{ij} \in \{0,1\}, && i \in N, j \in N,
\end{aligned} \tag{1}$$

Where y_i is a binary variable indicating whether bin i contains pieces, x_{ij} indicates whether piece j is packed into bin i, and n is the number of available bins (and also the number of pieces as we know we can pack n pieces into n bins).

A practical application of the one dimensional bin packing problem is cutting lengths of stock material that has fixed width, such as pipes for plumbing applications, or metal beams. A set of orders for different lengths must be fulfilled by cutting the stock lengths into smaller pieces while minimising the wasted material.

2 Initialisation of solutions

In this domain module, solutions are initialised by first randomising the order of the pieces, and then applying the 'first-fit' heuristic [2]. This is a constructive heuristic, which packs the pieces one at a time, each into the first bin that they will fit into.

3 Low Level Heuristics

Sections 3.1, 3.2, 3.3, and 3.4 explain the local search, mutational, ruin-recreate, and crossover low-level heuristics respectively. We have implemented seven low level heuristics in total, some of which are taken from [3].

3.1 Local Search Heuristics

These heuristics implement 'first-improvement' local search operators. In each iteration, a neighbour is generated, and it is accepted immediately if it has superior or equal fitness. If the neighbour is worse, then the change is not accepted. The behaviour of these heuristics is controlled with the 'depth of search' parameter. At the default value of 0.2, these heuristics iterate 10 times. If it is set higher, the heuristics iterate up to 20 times. Local search heuristics cannot produce a solution of worse fitness.

3.1.1 Swap

Select two different pieces at random, and swap them if there is space, and if it will produce an improvement in fitness.

3.1.2 Swap from Lowest Bin

Take the largest piece from the lowest filled bin, and exchange with a smaller piece from a randomly selected bin. If there is no such piece that produces a valid packing after the swap, then exchange the first piece with *two* pieces that have a smaller total size. If there are no such pieces then the heuristic does nothing.

3.2 Mutational Heuristics

The behaviour of these heuristics is controlled with the 'intensity of mutation' parameter. At the default value of 0.2, these heuristics run once. If it is set higher, the heuristics repeat up to five times, meaning a greater mutation is performed.

3.2.1 Swap

Select two different pieces at random, and swap them if there is space. If one of the pieces does not fit into the new bin then put it into an empty bin.

3.2.2 Split a Bin

This heuristic selects a bin at random from those with more pieces than the average. It then splits this bin into two bins, each containing half of the pieces from the original bin.

3.2.3 Repack the Lowest Filled Bin

Remove all of the pieces from the lowest filled bin, and repack them into the other bins if possible, with the best-fit heuristic.

3.3 Ruin-Recreate Heuristics

The behaviour of these heuristics is controlled by the parameter 'x'. At the default 'intensity of mutation' value of 0.2, x is set to three. If it is set higher, the value of x increases to fifteen. $x = 3$ when 'intensity of mutation' is between 0 and 0.2, $x = 6$ when 'intensity of mutation' is between 0.2 and 0.4, and so on, with x increasing by three at each interval.

3.3.1 Destroy x Highest Bins

Remove all the pieces from the x highest filled bins, where x is an integer determined by the 'intensity of mutation' parameter. Repack the pieces using the best-fit heuristic.

3.3.2 Destroy x Lowest Bins

Remove all the pieces from the x lowest filled bins, where x is an integer determined by the 'intensity of mutation' parameter. Repack the pieces using the best-fit heuristic.

3.4 Crossover Heuristics

3.4.1 Exon Shuffling Crossover

This crossover is detailed in [4].

4 Problem Instances

The instances of this class are taken from the literature. Interesting instances are selected from the following sources [5, 6, 7, 8]. Some instances are generated using the same methods as explained in those sources. Table 1 summarises the characteristics of the instances.

Set Name	Piece Sizes	Bin Capacity	Number of Pieces
bp1	Uniform [20,100]	150	1000
bp2	Triples [25,50]	100	2004
bp3	Triples [25,50]	100	1002
bp4	1-700	1000	160
bp5	Uniform [10,30]	100	2000
bp7	Bimodal Distribution: Half pieces from Gaussian (50,5) Half of pieces from Gaussian (35,5)	100	5000
bp8	Bimodal Distribution: Half pieces from Gaussian (50,5) Half of pieces from Gaussian (20,5)	100	5000

Table 1: The available instance sets

5 Fitness Function

A solution is given a fitness calculated from equation 2, where: n = number of bins, $fullness_i$ = sum of all the pieces in bin i, and C = bin capacity. The function puts a premium on bins that are filled completely, or nearly so. It returns a value between zero and one, where lower is better, and a set of completely full bins would return a value of zero.

$$Fitness = 1 - \left(\frac{\sum_{i=1}^{n} (fullness_i / C)^2}{n} \right) \tag{2}$$

References

[1] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, Chichester, 1990.

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

[3] Ruibin Bai, Jacek Blazewicz, Edmund K. Burke, Graham Kendall, and Barry McCollum. A simulated annealing hyper-heuristic methodology for flexible decision support. Technical report, University of Nottingham, 2007.

[4] Philipp Rohlfshagen and John Bullinaria. A genetic algorithm with exon shuffling crossover for hard bin packing problems. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO'07)*, pages 1365–1371, London, U.K., 2007.

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

[6] P. Schwerin and G. Wäscher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4(5):377–389, 1997.

[7] E. K. Burke, M. Hyde, and G. Kendall. Providing a memory mechanism to enhance the evolutionary design of heuristics. In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI'10)*, pages 3883–3890, Barcelona, Spain, July 2010.

[8] 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 ACM Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1559–1565, London, UK., July 2007.