PERGAMON

Computers & Operations Research 29 (2002) 821–839

computers & operations research

www.elsevier.com/locate/dsw

New heuristics for one-dimensional bin-packing

Krzysztof Fleszar[1], Khalil S. Hindi*,[2]

Department of Systems Engineering, Brunel University, Uxbridge, Middlesex UB8 3PH, UK

Received 1 February 2000; received in revised form 1 August 2000

Abstract

Several new heuristics for solving the one-dimensional bin packing problem are presented. Some of these are based on the minimal bin slack (MBS) heuristic of Gupta and Ho. A different algorithm is one based on the variable neighbourhood search metaheuristic. The most effective algorithm turned out to be one based on running one of the former to provide an initial solution for the latter. When tested on 1370 benchmark test problem instances from two sources, this last hybrid algorithm proved capable of achieving the optimal solution for 1329, and could find for 4 instances solutions better than the best known. This is remarkable performance when set against other methods, both heuristic and optimum seeking.

Scope and purpose

Packing items into boxes or bins is a task that occurs frequently in distribution and production. A large variety of different packing problems can be distinguished, depending on the size and shape of the items, as well as on the form and capacity of the bins (H. Dyckhoff and U. Finke, Cutting and Packing in Production and Distribution: a Typology and Bibliography, Springer, Berlin, 1992). Similar problems occur in minimising material wastage while cutting pieces into particular smaller ones and in the scheduling of identical processors in order to minimise total completion time. This work addresses the basic packing problem, known as the one-dimensional bin packing problem, where it is required to pack a number of items into the smallest possible number of bins of pre-specified equal capacity. Even though this problem is simple to state, it is NP hard, i.e., it is unlikely that there exists an algorithm that could solve every instance of it in polynomial time. Solution of more general realistic packing problems is probably contingent upon the availability of effective and computationally efficient solution procedures for the basic problem. In this work we present several heuristics capable of doing that. Extensive computational testing attests to the power of these heuristics, as well as to their computational efficiency. © 2002 Published by Elsevier Science Ltd.

Keywords: Bin packing; Variable neighbourhood search; Heuristics

* Corresponding author. Tel.: + 44-(0)-1895-203299; fax: + 44-(0)-1895-812556.
E-mail address: khalil.hindi@brunel.ac.uk (K.S. Hindi).

[1] On leave from the Faculty of Electronics and Information Technology of Warsaw University of Technology.

[2] http://www.brunel.ac.uk/ ~ emstksh

0305-0548/02/$ - see front matter
PII: S0305-0548(00)00082-4

1. Introduction

Consider the following problem: given a finite set O of numbers and two constants c and m, does there exist a partition of O into m or less subsets, such that the sum of elements in any of the subsets does not exceed c?

This is a NP-complete decision problem [1]. It naturally gives rise to an associated NP-hard optimisation problem: what is the minimum number of subsets necessary to effect the partition of O under the condition that the sum of elements of any of the subsets does not exceed c? This problem, which is considered in this paper, can be stated in the following terms: given a number of items $i = 1, \ldots, n$ of integer sizes t_i, what is the minimum number of bins, each having the same integer capacity c, necessary to pack all items; hence the name bin packing problem (BPP).

Since the BPP is NP-hard in the strong sense [1], it is unlikely that there exists a polynomial time algorithm to solve it optimally. Scholl et al. [2] offer a good survey of existing solution procedures, as well as a good exact algorithm that they have developed. Two subsequent works are noteworthy. The first is an exact algorithm by Carvalho [3], based on column generation and branch-and-bound, and the second is a heuristic, named the minimum bin slack heuristic, by Gupta and Ho [4].

This paper presents some new heuristic algorithms. It will be assumed throughout that items are sorted in a non-increasing order of their sizes and stored in a list Z; a task that can be accomplished in $O(n \log n)$.

Table 1
Symbols

Symbol	Description
c	bin capacity
n	number of items
i, j	item number
t_i	size of item i
$t^{\min}$	smallest item size
p_i	probability of selecting item i
m, m^*	number of bins, optimal number of bins
α, β	bin number
Z, Z'	list of items
q, r	list index
Z_q, Z'_q	qth item on the list
$\mathscr{A}_\alpha$	set of items assigned to bin α
$\mathscr{A}$, $\mathscr{A}^*$	set of items
b_i	number of bin to which item i is assigned
$l(\alpha)$	load of bin α, i.e. sum of the sizes of items assigned to bin α
$s(\alpha)$	free space in bin α, i.e., $c - l(\alpha)$
$s(\mathscr{A})$	slack resulting from assigning items in $\mathscr{A}$ to a bin
x, x', x''	solution, i.e. complete assignment of items to bins
k, $k^{\max}$	neighbourhood number, maximum neighbourhood size
$N_k(x)$	set of solutions in kth neighbourhood of x
f, Δf	objective function, change in objective function

The major symbols used in the presentation are given in Table 1. Since most of the heuristics presented here are improving heuristics, a lower bound is used as one of the stopping criteria, i.e. if the number of bins in the current incumbent achieves the value of the lower bound, then the processing is stopped. Apart from the simple lower bound $\lceil (1/c)\sum_{i=1}^{n} t_i \rceil$, some lower bounds based on dual feasible functions developed by Fekete and Schepers [5] have been used.

2. Minimum bin slack

The minimum bin slack (MBS) heuristic of Gupta and Ho [4] is bin-focused. At each step, an attempt is made to find a set of items (packing) that fits the bin capacity as much as possible. In this sense, MBS is similar to Hoffmann's algorithm [6] for solving assembly line balancing problems.

At each stage, a list Z' of n' items not assigned to bins so far, sorted in the non-increasing order of sizes is kept. Each time a packing is determined, the items involved are placed in a bin and removed from Z', preserving the sort order. The process ends when the list becomes empty.

Each packing is determined in a search procedure that tests all possible subsets of items on the list which fit the bin capacity. The subset that leaves the least slack is adopted. If the algorithm finds a subset that fills the bin up completely, the search is stopped, for there is no better packing possible. The search is started from items of a greater size, i.e., from the beginning of Z', because items of relatively big sizes are usually harder to pack in bins and, therefore, an attempt to pack them first should be undertaken.

Building a packing for one bin can be implemented iteratively or recursively. A recursive implementation is shown in Fig. 1. The procedure is invoked with $q = 1$ and $\mathscr{A} = \mathscr{A}^* = \emptyset$, where q is the index of the item in Z' from which the processing is started, $\mathscr{A}$ is the set of items currently assigned to the packing, and $\mathscr{A}^*$ is the set of items in the best packing. The slack in packing $\mathscr{A}$ is expressed by $s(\mathscr{A})$ and the number of items in Z' is denoted by n'. Clearly, $s(\mathscr{A})$ can be computed simply by starting from $s(\mathscr{A}) = c$ and updating every time an item is added to or removed from $\mathscr{A}$.

Gupta and Ho [4] show that MBS is superior, in terms of solution quality, to the widely used first-fit decreasing (FFD) and the best-fit decreasing (BFD) algorithms [7]. It also finds an optimal solution for problem instances for which a two-bin solution exists.

```
procedure MBSOnePackingSearch(q)
begin
    for r := q to n' do
    begin
        i := Z'_r;
        if t_i ≤ s(A) then
        begin
            A := A ∪ {i};
            MBSOnePackingSearch(r + 1);
            A := A \ {i};
            if s(A*) = 0 then Exit;
        end;
    end
    if s(A) < s(A*) then A* := A;
end;
```

Fig. 1. MBS one-packing search procedure.

Although the complexity of this procedure is $O(2^n)$, computational experience shows that it is quite efficient in solving practical problem instances with various parameters. This is due to the fact that in practical problem instances, the number of items involved in each packing is much smaller than n. If the maximum number of items that can be placed in one bin is u, then the complexity of MBS-one-packing procedure is reduced to $O(n^u)$ and thus packings for all bins are built in $O(n^{u+1})$. This is why MBS, which is a non-polynomial algorithm, gives solutions in a reasonable time. Nevertheless, it is easy to create some small instances for which MBS will spend hours before finding a solution. For example, it is enough to have the bin capacity equal to an odd number and have all item sizes equal to even numbers in order to force the algorithm to search through all possible packings, since no full packing (i.e., a packing with zero slack) exists in this case.

We propose introducing some additional conditions in order to save some computation:

1. In the 'for' loop, if $r > q$ and the sizes of items Z'_{r-1} and Z'_r are the same, then processing for item Z'_r is skipped. This is induced by the fact that considering item Z'_r will not generate packings other than those generated earlier with item Z'_{r-1}.
2. Let T_r be the sum of the sizes of items $Z'_r, \ldots, Z'_{n'}$. If, in the 'for' loop, $s(\mathscr{A}) - T_r \geqslant s(\mathscr{A}^*)$, then the procedure is exited, for it is not possible to build a packing better than $\mathscr{A}^*$ with the items left. This can save some considerable computation when the bin size is very large in comparison to item sizes. Note that T_r does not have to be recalculated each time; it is sufficient to start from the sum of sizes of all items and update the value during computation.
3. At the beginning of the procedure, if $s(\mathscr{A}) < t^{\min}$, where $t^{\min}$ is the size of the smallest item, then the 'for' loop is skipped, since no item can be added to packing $\mathscr{A}$.

The modifications presented above save some computation, but do not influence the final solution given by the algorithm. In the following section, we introduce some new heuristics that are based on MBS, but may give different results.

3. New MBS-based heuristics

In this section four new heuristics are presented. The first, MBS′, is a slightly modified version of the original algorithm, that, however, leads to different solutions. The other three heuristics are based on MBS′.

3.1. MBS′

The following modification to MBS is proposed: before the one-packing search procedure is invoked, an item (the seed) is chosen and permanently fixed in the packing. This can be done because every item must be placed in a bin anyway. A good choice of seed is the item of the greatest size, i.e., the first on the list Z'. This will leave the least space in the bin to fill during the search, thereby shortening the time of processing. Moreover, the solution process will be forced to use larger, and for that reason more trouble-causing, items first.

The modified algorithm, which we will refer to as MBS′, creates packings for the successive bins in the same manner as the original algorithm, using the same one-packing search procedure shown

in Fig. 1. The only difference is the initiation of the search procedure; since the first item is always fixed permanently in packing $\mathscr{A}$, the procedure is invoked with $q = 2$ and $\mathscr{A} = \mathscr{A}^* = \{Z'_1\}$.

Obviously, MBS′ may give different packings and thus different solutions from those achieved by the original algorithm. It may obtain inferior solutions, if, for example, it creates some non-full packings for some of the first bins, involving items that could take part in many full packings if MBS were used. However, the new algorithm may lead to superior results. This is due to the fact that MBS tends to build packings using small items, leaving till later big ones for which a full packing cannot be found, with the result that the search may end up with bins having a large slack. To illustrate, consider a very simple example, in which 3 items of sizes $t_1 = t_2 = t_3 = 5$ and 3 items of sizes $t_4 = t_5 = t_6 = 3$ are to be packed into bins of size $c = 9$. When MBS is applied, a full packing (4, 5, 6) is built for the first bin. Since the size of every item left is greater than half the bin capacity, 3 more bins are needed, resulting in a 4-bin solution. However, MBS′ fixes the first item in the first packing. As a result, one item of size 5 and one item of size 3 are put in the first bin and two more packings are created in the same manner, to give a 3-bins solution.

In summary, MBS′ should be faster than MBS. In addition, although the former does not dominate the latter in terms of solution quality, it should empirically do better on average. Some of the computational experiments, discussed in Section 6, are designed to test this conjecture.

3.2. Relaxed MBS′

This heuristic, along with both heuristics described in the following two subsections, is based on MBS′. The modification proposed here is to accept some packings that leave positive slack, without seeking better ones. This can be done by changing the stopping condition in the one-packing search procedure from $s(\mathscr{A}^*) = 0$ to $s(\mathscr{A}^*) \leqslant$ allowable slack. Note, however, that the procedure accepts maximal packings only, i.e., those packings that cannot be enlarged by the addition of any item left, without violating the capacity limit.

After some experimentation, the following scheme proved most effective: in addition to running the procedure with zero allowable slack (i.e., MBS′), it is run several more times, such that the level of allowable slack is increased by $v = \lceil 0.5\% \times c \rceil$ each time and the number of steps limited to $\min\{40, c/v\}$. Naturally, processing is stopped earlier if the number of bins in the current incumbent is equal to the lower bound value.

3.3. Perturbation MBS′

Starting from an initial solution, which is not necessarily an MBS′ solution, this heuristic successively builds a new solution by perturbing the current one. The perturbation is effected in the following way: an item from a bin with a relatively large slack is selected as a seed and a packing containing this item, considering all other items, is created employing the one-packing search procedure. Items corresponding to the new packing are transferred to a new bin, which is added to the solution. If during this process a bin becomes empty, it is immediately removed from the solution.

The rationale is to force the one-packing search procedure to create a new packing using items from worse-filled bins rather than from better-filled bins. To accomplish this aim, however, seed selection and list ordering have to be carefully defined:

Seed selection: The seed is selected at random with the probability of choosing item i equal to

$$p_i = \frac{s(b_i)}{\sum_{j=1}^{n} s(b_j)},$$

where b_i is the bin in which item i currently resides. Consequently, items from full bins ($s(b_i) = 0$) are not selected as seeds and the probability of selection grows with the amount of slack in the corresponding bin.

List order: After the seed is selected, it is placed at the beginning of the list Z' and all other items are stored in the decreasing order of the slacks in their corresponding bins, with ties broken arbitrarily. To speed up the sorting, items on the list are divided first in $O(n)$ into two parts: items Z'_2 to Z'_q with slack in the corresponding bins greater than zero, and Z'_{q+1} to Z'_n with slack in their bins equal to zero. Only items Z'_2 to Z'_q need then to be sorted in $O(q \log q)$. This saves some computation, since in a suboptimal solution many bins are full and very few items are placed in the first part of the list.

Both seed selection and item ordering give preference to using items that reside in less full bins and tend to save the full and nearly full bins in the current solution. Moreover, randomisation of these two procedures serves at one and the same time to avoid cycling and diversify the search.

After each step is performed, i.e., after a new packing is created and placed in a new bin and any empty bin is deleted, a new solution is compared to the current incumbent: if it has fewer bins, then it is adopted. Processing is stopped when the procedure fails to improve the incumbent for a certain number of steps or when the lower bound on the number of bins is achieved.

It is worth noting that not every step of this heuristic improves the solution in terms of the number of bins. However, the long-term trend is in fact improving, since the method builds promising packings, and the final solution is often better than the initial.

3.4. Sampling MBS′

This algorithm invokes MBS′ several times, changing the ordering of items in list Z' each time, and adopts the best solution. Processing is stopped when the lower bound on the number of bins is achieved or when the algorithm fails to improve the incumbent for a certain number of passes. In this sense, the procedure is similar to the sampling methods developed by many authors for the resource-constrained project scheduling problem [8].

The order in Z' is based probabilistically on the non-increasing order of item sizes, with the probability of selecting item i at each step of filling the order vector, Z', equal to

$$p_i = \frac{t_i^2}{\sum_{j \in \mathscr{A}} t_j^2},$$

where $\mathscr{A}$ is the set of items not yet selected. Once Z' is completed, MBS′ is run.

4. Variable neighbourhood search

Local search methods for combinatorial optimisation proceed from an initial solution by performing a sequence of local changes, each improving the value of the objective function, until

a local optimum is found. Systematic change of neighbourhood within a local search algorithm yields a simple and effective metaheuristic, called the variable neighbourhood search (VNS), which is due to Hansen and Mladenovic [9,10]. This search scheme explores increasingly distant neighbourhoods of the current incumbent solution and jumps from there to a new one if and only if an improvement has been made.

A finite set of neighbourhood structures N_k ($k = 1, \ldots, k^{\max}$) that will be used in the search is defined, with $N_k(x)$ being the set of solutions in the kth neighbourhood of x. Moreover, an initial solution x is provided. This can be obtained from any simple heuristic. In the main loop of VNS, the following three steps are started from $k = 1$ and repeated until $k > k^{\max}$:

Shaking: A point x' is generated at random from the kth neighbourhood of x, i.e., from $N_k(x)$.

Local search: A local search method is applied with x' as the initial solution to obtain a local optimum x''.

Move or not: If the local optimum is better than the incumbent, then it is accepted, i.e., $x := x''$, and the processing is restarted by assigning $k := 1$. Otherwise, the neighbourhood is increased, i.e., $k := k + 1$.

Point x' is generated at random, in order to avoid cycling, which might occur if a deterministic rule were used.

Several variations of VNS are possible. The *local search* step can be carried out by steepest descent, first improving move, or any descent algorithm in between. It can also be carried out, for large problem instances, over a subset of solution attributes and these can change dynamically. The *shaking* and *local search* steps can be repeated several times and the best solution achieved in this way adopted. This amounts to carrying out more detailed search (intensification) in the immediate neighbourhood of the incumbent. Another possibility is to move to the local optimum after the *local search* step whether it is better than the incumbent or not, reverting to the first neighbourhood ($k := 1$) if it is better and enlarging the neighbourhood ($k := k + 1$) otherwise. This should help escape local optima in problems with an awkward search space.

5. VNS algorithm for the BPP

The variable neighbourhood search algorithm for the BPP is based on *moves*. A move is defined as the transfer of an item from its current bin to another or the swap of a pair of items across their respective current bins. Only valid moves, i.e., moves that do not violate the bin capacity constraint, are considered. This is the same move strategy used in [2] in the context of tabu search.

5.1. Objective function

The crucial factor of any neighbourhood search procedure is the choice of the objective function. In the present context, choosing the real objective function, which is to minimise the number of bins, is rather pointless, since very many different configurations, in terms of the assignment of items to bins, correspond to the same number of bins. Observing that a good solution will always have nearly full bins leads naturally to an objective function that seeks configurations having this feature. One such objective function is

$$\max f(x) = \sum_{\alpha=1}^{m} (l(\alpha))^2, \tag{1}$$

where m is a number of bins in x and $l(\alpha)$ denotes a sum of sizes of items $\mathscr{A}_\alpha$ assigned to bin α, i.e., $l(\alpha) = \sum_{i \in \mathscr{A}_\alpha} t_i$.

Along with maximising bin loads, the objective function seeks to reduce the number of bins. It is also worth observing that the value of the function will not change if an empty bin is added or removed.

5.2. Shaking

The kth neighbourhood of solution $x(N_k(x))$ is defined as the set of solutions that can be obtained from x by successively performing k moves of some distinct items. To find a random solution in $N_k(x)$, k random moves are performed on x. A random move is generated as follows:

1. A list, Z', is created by copying from Z the items that have not been moved so far, preserving the non-increasing item size order.
2. A random item i is selected from list Z'.
3. All possible moves involving i are determined and saved:
   (a) Transfers are determined by considering all bins $\alpha = 1, \ldots, m$, where m is the number of bins in x. The transfer of item i to bin α is saved only if $t_i \leqslant s(\alpha)$, i.e., if there is enough space in bin α for item i ($s(\alpha) = c - l(\alpha)$).
   (b) Swaps are determined by considering the items in the list Z' other than i. Processing starts from the end of the list and finishes in one of two cases: either the beginning of the list is reached or item $j = Z'_q$ is too big to be exchanged with item i, i.e., $t_j > t_i + s(b_i)$. In the latter case, all items in front of q in Z' are not considered since they are not smaller than j and cannot, therefore, be swapped with i.
   Swap $i \leftrightarrow j$ is saved only if two conditions are satisfied: $t_i \neq t_j$ (swapping items of the same size does not change the solution) and $t_i \leqslant t_j + s(b_j)$, i.e., there is enough space for i to place it in bin b_j after removing j (the opposite condition is ensured by the stopping condition).
4. If there is no possible move for item i, then i is removed from Z' and processing is restarted from step 2. Otherwise, from all possible moves involving i, a random move is selected and performed. Item i and the counterpart item in the case of a swap are marked as moved, in order to exclude their participation in further moves in the current shaking.

This process is repeated until k moves are performed or until it is not possible to perform any move using items not marked as moved. Note that if there are no possible moves for an item, i, it is removed from Z' in step 4 above only temporarily, since after some transfers or swaps are performed, moves involving i can be found.

An important feature of this shaking procedure is that it always preserves the number of bins. An advantage of this is that a solution with a larger number of bins will not be created. On the other hand, it is not certain that every optimal solution can be obtained starting from a given solution by shaking in this manner.

The complexity of the shaking procedure is $O(kn^2)$. However, if the conditional jump in step 4 is omitted, the complexity decreases to $O(kn)$. In fact, the jump is almost always omitted, because even if the overall slack in the current solution is very small and there are very few possible transfers, there are still many possible swaps normally. Moreover, a great deal of computation is saved owing to the additional stopping conditions based on having the list of items, Z', sorted in a non-increasing order of sizes.

5.3. Local search

A local optimum is determined by a steepest descent algorithm. At each step, the procedure effectively enumerates all possible improving moves (transfers and swaps) and performs one that maximises the improvement of the objective function (1), if any. There is no need in this scheme to recalculate the objective function from scratch when evaluating a move, but merely the change in the objective function that would result. Thus, the value of a transfer of item i from bin α to bin β is calculated as

$$\Delta f = [l(\alpha) - t_i]^2 + [l(\beta) + t_i]^2 - (l(\alpha))^2 - (l(\beta))^2. \tag{2}$$

Analogously, the value of swapping items i and j across their respective bins α and β is

$$\Delta f = [l(\alpha) - t_i + t_j]^2 + [l(\beta) + t_i - t_j]^2 - (l(\alpha))^2 - (l(\beta))^2. \tag{3}$$

Since moves involving items from full bins do not increase the objective function, these are excluded from consideration. Thus at the beginning of each pass of the local search procedure, a list Z' of all items i for which $l(b_i) < c$ is created in a linear time, by copying from Z while preserving order. Thereafter, the following is carried out:

Transfers: For each non-full bin α, items are taken from Z' starting from the end of the list. For each item i, the value of transferring it from b_i to α ($b_i \neq \alpha$) is computed (Eq. (2)) and the best transfer is saved. The evaluation is stopped either when the beginning of the list is reached or when the item $j = Z'_q$ is too big to fit into bin α, i.e., when $t_j > s(\alpha)$. In the latter case all items in Z' in front of q are not considered, since they are not smaller than j and cannot, therefore, be put in α.

Swaps: Items from list Z' are considered. For each item $i = Z'_q$ the following is carried out:

1. $r = q - 1$.
2. While item $j = Z'_r$ has the same size as i, i.e., while $t_j = t_i$, decrease r (swapping items of the same size does not change the solution).
3. For each item in Z' starting from r backwards, the value of swapping i with $j = Z'_r$ is computed (Eq. (3)) and the best swap is saved. Processing is terminated either when the beginning of the list is reached or when j is too big to be exchanged with i, i.e., when $t_j > t_i + s(b_i)$. Note that there is always space for item i in bin b_j, since i is always follows j in Z' ensuring that $t_i \leqslant t_j$.

Observe that in a near-optimal solution, many bins are full. Consequently, the number of items in Z' is usually small and there are very few bins to which items can be transferred.

Naturally, since the overarching goal of the local search is to reduce the number of bins, once a bin becomes empty, it is immediately removed from the solution.

5.4. Overall VNS scheme

The VNS algorithm for BPP follows the general description in Section 4. The MBS′ heuristic (Section 3.1) is used to find an initial solution. Starting from solutions provided by other simple heuristics, like the first fit decreasing and the best fit decreasing [7] was also experimented with, but the overall results proved to be inferior.

The shaking and the local search procedures are performed as described in Sections 5.2 and 5.3, respectively. When a new local optimum x'' is compared to the current incumbent x, the numbers of

bins $m(x'')$ and $m(x)$ are compared first. If $m(x'') < m(x)$ then the new solution is adopted, i.e. $x := x''$; otherwise, if $m(x'') = m(x)$ we proceed to evaluate $f(x'')$ and if $f(x'') > f(x)$, the new solution is also adopted.

The search is stopped either when the lower bound on the number of bins is achieved indicating that the solution is optimal, or when $k > k^{\max}$. Choosing an appropriate value of $k^{\max}$ is a question of balance: a large value is likely to lead to some improvement in the quality of the results, but increases computation time. $k^{\max} = 20$ was found by experimentation to lead to reasonably short processing times, without significantly compromising the quality of the overall solutions. Moreover, increasing this value further did not give a significant improvement. Nor did relating it to problem instance size.

6. Computational analysis

The algorithm was coded in Borland Delphi Pascal version 4 and all the tests were performed on a 400 MHz, Pentium II PC, running under Windows NT 4.0.

6.1. Data sets

Two classes of benchmark bin packing problem instances available in the OR Library (http://www.ms.ic.ac.uk/info.html) were used [11]. The first class (U class) consists of items of sizes uniformly distributed in (20, 100) to be packed into bins of size 150. There are four sets in this class, namely U120, U250, U500 and U1000. Each consists of 20 instances with 120, 250, 500 and 1000 items, respectively.

The second class (T class) consists of 'triplets' of items from (25, 50) to be packed into bins of size 100. There are also four sets in this class, namely T60, T120, T249 and T501. Each set contains 20 instances with 60, 120, 249 and 501 items, respectively.

All problem instances in both the U class and T class have been solved to optimality with an exact algorithm by Carvalho [3]. For the T class, all the instances are designed such that in the optimal solution each bin is filled up with 3 items. Thus the number of bins in the globally optimal solution for each instance is equal to the number of items divided by 3. The difficulty of the T class is caused by the lack of any slack in the optimal solution.

A third class of benchmark bin packing problem instances (B class) is available from http://www.bwl.tu-darmstadt.de/bwl3/forsch/projekte/binpp/. This class contains three sets, which for the purposes of this article are called B1, B2 and B3. The B1 set consists of 720 instances, of which 704 are currently solved optimally. The B2 set consists of 480 instances, the optimal solutions for 477 of which are known. In both the B1 and B2 sets, the number of items varies from 50 to 500. The B3 set consists of 10 instances, each of 200 items of a size uniformly distributed in (20 000, 35 000) to be packed into bins of size 100 000. This set is considered to be very hard. The optimal solutions of only 3 of the B3 instances are known. Detailed description of all parameters for the three sets of the B class can be found in [2].

Results are compared to the reference solutions, which are the optimal solutions or the best-known lower bounds where the optimum is not known (26 cases out of 1370). The absolute deviation is represented by the number of bins and the relative deviation is computed as

the absolute deviation divided by the number of bins in the optimal solution (or in the best-known lower bound). The time of processing is measured in seconds.

6.2. MBS vs. MBS′

Tables 2 and 3 present the results of MBS and MBS′ heuristics. N indicates the number of instances in each test set. The 'hits' column shows the number of the reference solutions achieved. The next four columns show the absolute deviation (abs. dev.) and the relative deviation (rel. dev.) from the reference solution. For both, the average (av.) and the maximum (max.) values over all members of each set are displayed. The average time and maximum time of processing over all members of each set are also shown, in seconds.

For the U class both algorithms achieve similar results. In comparison to MBS, MBS′ achieves one inferior solution in U120, but two better solutions in U250. This shows that, as conjectured, neither dominates the other.

None of the 'triplet' instances was solved by either of the two heuristics, but MBS achieves for the bigger T instances a noticeably smaller average and maximum deviations in comparison with MBS′.

MBS′ does well on the B class problem instances; it achieves the reference solution in more than 87% of the B1 instances, compared to only 35% for MBS. However, the deviation of the MBS′ results is still worse than that of MBS for B2 and B3.

Although the number of items in the larger problem instances reaches 1000, processing time is surprisingly short. Both heuristics process all U, T and B1 instances in no more then 0.05 s per instance. Differences in processing time become noticeable for the B2 set, which is processed by MBS′ twice faster on average than by MBS. The contrast is even clearer for the B3 instances, for which MBS′ is more than ten times faster than MBS.

Table 2
MBS

Set	N	Hits	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	20	12	0.50	2	1.02	4.17	0.00	0.01
U250	20	10	0.70	3	0.68	2.91	0.00	0.01
U500	20	11	0.60	2	0.30	1.01	0.01	0.01
U1000	20	7	0.80	3	0.20	0.73	0.02	0.02
T60	20	0	1.00	1	5.00	5.00	0.00	0.00
T120	20	0	1.00	1	2.50	2.50	0.00	0.01
T249	20	0	1.15	2	1.39	2.41	0.01	0.01
T501	20	0	2.05	3	1.23	1.80	0.02	0.04
B1	720	251	1.49	9	1.89	16.67	0.01	0.05
B2	480	387	0.28	5	0.58	14.29	0.11	8.99
B3	10	0	3.30	4	5.95	7.27	0.77	1.03
ALL	1370	678	1.02	9	1.42	16.67	0.05	8.99

Table 3
MBS′

Set	N	Hits	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	20	11	0.45	1	0.91	2.17	0.00	0.01
U250	20	12	0.45	2	0.44	1.90	0.00	0.01
U500	20	11	0.60	2	0.30	1.01	0.01	0.01
U1000	20	7	0.80	3	0.20	0.73	0.02	0.03
T60	20	0	1.00	1	5.00	5.00	0.00	0.01
T120	20	0	1.00	1	2.50	2.50	0.00	0.01
T249	20	0	1.80	3	2.17	3.61	0.00	0.01
T501	20	0	3.80	7	2.28	4.19	0.01	0.02
B1	720	629	0.17	3	0.23	5.26	0.00	0.02
B2	480	381	0.35	9	0.69	14.29	0.06	4.57
B3	10	0	4.00	5	7.21	9.09	0.07	0.08
ALL	1370	1051	0.38	9	0.61	14.29	0.02	4.57

Overall MBS′ achieves the reference solutions for many more instances than MBS and gives a smaller average deviation. Moreover, processing times for MBS′ are shorter. Therefore, for all the following algorithms, MBS′ is used to obtain an initial solution.

6.3. *MBS′-based heuristics*

Since all these heuristics start from an initial solution provided by MBS′, their results are presented in terms of improvement over the initial solution. Thus '+hits' records the number of instances for which the reference solution was obtained by the heuristic under consideration but had not been obtained by MBS′, i.e., the number of the reference solutions achieved excluding those achieved by MBS′. The number of instances not solved to the reference value is shown in the 'inferior' column. Note that the average and maximum deviations were calculated over all instances, not just those not solved by MBS′. Also the average and maximum computation times include MBS′.

6.3.1. *Relaxed MBS′*

The relaxed MBS′ heuristic runs MBS′ 41 times with slack allowed in bins varying from 0% to 20% in steps of 0.5% of bin capacity (see Section 3.2). The results are shown in Table 4. The heuristic performs quite well for all 'uniform' instances, especially the small ones. It is remarkable that it achieves 9 of the 10 best-known solutions for the B3 instances, which, due to their parameters (bin size $c = 100\,000$, item sizes from 20 000 to 35 000), are known to be very hard. Unfortunately, relaxed MBS′ does not solve any 'triplet' instances. However, in comparison with MBS′, the average deviation is much smaller.

Table 4
Relaxed MBS′

Set	+ Hits	Inferior	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	+ 8	1	0.05	1	0.10	2.00	0.00	0.01
U250	+ 4	4	0.20	1	0.19	0.97	0.01	0.04
U500	+ 5	4	0.20	1	0.10	0.51	0.03	0.13
U1000	+ 6	7	0.35	1	0.09	0.25	0.19	0.48
T60	0	20	1.00	1	5.00	5.00	0.01	0.01
T120	0	20	1.00	1	2.50	2.50	0.02	0.02
T249	0	20	1.00	1	1.20	1.20	0.08	0.09
T501	0	20	1.25	2	0.75	1.20	0.32	0.36
B1	+ 52	39	0.07	2	0.05	2.44	0.04	0.21
B2	+ 44	55	0.14	6	0.36	14.29	0.18	5.88
B3	+ 2	8	0.80	1	1.44	1.85	0.53	0.65
ALL	+ 121	198	0.16	6	0.31	14.29	0.09	5.88

6.3.2. *Perturbation MBS′*

Table 5 shows the results of the Perturbation MBS′ heuristic. The number of unsuccessful steps after which algorithm is stopped has been set to 1000, on the basis of some preliminary experimentation.

The algorithm is very well suited to solving instances for which bin loads are completely or nearly full in the optimal solution. Although this heuristic gives relatively poor results for the 'uniform' instances, it solves all 'triplet' instances to optimality; in fact, it is the only heuristic presented in this paper that solves any of 'triplet' instances at all. It is also remarkable, that it finds solutions for each of these instances in no more than 0.1 of a second.

6.3.3. *Sampling MBS′*

The results of the sampling MBS′ heuristic are shown in Table 6. The number of unsuccessful steps, after which the algorithm is stopped has been set to 50 on the basis of some preliminary experimentation.

In comparison with the algorithms presented before it, sampling MBS′ deals quite well with the U class and the B class. It has the noticeable advantage of giving comparatively small average and maximum deviations from the best-known solutions. For all the T instances, the solutions obtained are worse than the optimal by only one bin and the number of bins in all other instances is never greater than the reference by more than two.

6.4. *The VNS algorithm*

Table 7 shows the results for the VNS algorithm. It works remarkably well for all 'uniform' instances. The reference solution is not achieved for only 2 instances in the U class and 40 instances

Table 5
Perturbation MBS′

Set	+ Hits	Inferior	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	0	9	0.45	1	0.91	2.17	0.01	0.04
U250	+ 1	7	0.40	2	0.39	1.90	0.02	0.06
U500	+ 3	6	0.30	1	0.15	0.51	0.04	0.14
U1000	+ 9	4	0.20	1	0.05	0.25	0.07	0.25
T60	+ 20	0	0.00	0	0.00	0.00	0.01	0.02
T120	+ 20	0	0.00	0	0.00	0.00	0.02	0.04
T249	+ 20	0	0.00	0	0.00	0.00	0.02	0.04
T501	+ 20	0	0.00	0	0.00	0.00	0.06	0.10
B1	+ 8	83	0.15	3	0.21	5.26	0.06	0.65
B2	+ 24	75	0.25	5	0.56	14.29	0.09	4.57
B3	0	10	4.00	5	7.21	9.09	1.75	1.86
ALL	+ 125	194	0.22	5	0.38	14.29	0.08	4.57

Table 6
Sampling MBS′

Set	+Hits	Inferior	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	+8	1	0.05	1	0.10	2.00	0.00	0.03
U250	+4	4	0.20	1	0.19	0.97	0.03	0.13
U500	+6	3	0.15	1	0.07	0.49	0.06	0.33
U1000	+12	1	0.05	1	0.01	0.24	0.11	1.29
T60	0	20	1.00	1	5.00	5.00	0.02	0.02
T120	0	20	1.00	1	2.50	2.50	0.05	0.06
T249	0	20	1.00	1	1.20	1.20	0.13	0.14
T501	0	20	1.00	1	0.60	0.60	0.36	0.37
B1	+34	57	0.09	2	0.11	5.00	0.05	0.55
B2	+83	16	0.04	2	0.06	5.56	0.11	16.63
B3	0	10	1.90	2	3.43	3.70	2.43	2.82
ALL	+147	172	0.14	2	0.24	5.56	0.10	16.63

in the B class. It is, however, worth recalling that for 26 of these 40 instances, comparison is with the best-known lower bound since the optimal solutions are not known.

Similarly to the relaxed MBS′ and the sampling MBS′, VNS does not find the optimal solution of any of the 'triplet' instances. However, overall, the deviation from the best-known results is very small on average and never exceeds 1.

Table 7
VNS

Set	+hits	Inferior	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
U120	+9	0	0.00	0	0.00	0.00	0.00	0.02
U250	+7	1	0.05	1	0.05	0.95	0.01	0.11
U500	+8	1	0.05	1	0.02	0.48	0.02	0.10
U1000	+13	0	0.00	0	0.00	0.00	0.03	0.07
T60	0	20	1.00	1	5.00	5.00	0.04	0.06
T120	0	20	1.00	1	2.50	2.50	0.06	0.08
T249	0	20	1.00	1	1.20	1.20	0.08	0.16
T501	0	20	1.00	1	0.60	0.60	0.15	0.23
B1	+65	26	0.04	2	0.04	2.44	0.08	1.18
B2	+93	6	0.01	1	0.03	2.94	0.07	4.57
B3	+2	8	0.80	1	1.44	1.85	2.06	3.33
ALL	+197	122	0.09	2	0.18	5.00	0.09	4.57

As mentioned earlier, increasing $k^{\max}$ did not improve solutions significantly. For example, increasing it from 20 to 50 improved the solutions of only 3 instances, but with more than a three fold increase in average processing time.

6.5. Final algorithm

Recall that the perturbation MBS′, itself started from MBS′, is the only algorithm developed here that solves the 'triplet' instances, while VNS gives the best results for the 'uniform' instances. Consequently, the final algorithm has been constructed as follows: perturbation MBS′ is run first and, if it does not achieve the optimal number of bins, its solution is then used as the initial solution of the VNS algorithm. This combination proved to be the most effective among all the other possible combinations of the heuristics developed in this work.

The results of the final algorithm are shown in Table 8. Although the algorithm does not improve the starting solutions generated by MBS′ in approximately 77% of the cases, it, nevertheless, leads to remarkably good overall results.

For 4 problem instances, solutions better than the best known so far were found. Table 9 shows the values of these solutions, none of which, however, is proven to be optimal. In fact, all of the improved solutions are due to the VNS algorithm. One of the improved solutions was also achieved by both the sampling MBS′ and the relaxed MBS′ heuristics.

Finally, Table 10 compares the performance of all the algorithms developed in this work over all 1370 benchmark problem instances.

6.6. Comparison with existing methods

Comparison with the best results available in the literature is illuminating. Carvalho [3] solves all the U and T class problems to optimality. The biggest instances, i.e., the U1000 and T501, are

Table 8
Perturbation MBS′ + VNS

Set	N	hits	Inferior	Abs. dev.		Rel. dev.		Time	
				av.	max.	av.	max.	av.	max.
U120	20	20	0	0.00	0	0.00	0.00	0.02	0.04
U250	20	19	1	0.05	1	0.05	0.95	0.03	0.16
U500	20	20	0	0.00	0	0.00	0.00	0.04	0.14
U1000	20	20	0	0.00	0	0.00	0.00	0.07	0.27
T60	20	20	0	0.00	0	0.00	0.00	0.01	0.01
T120	20	20	0	0.00	0	0.00	0.00	0.02	0.04
T249	20	20	0	0.00	0	0.00	0.00	0.02	0.04
T501	20	20	0	0.00	0	0.00	0.00	0.06	0.10
B1	720	694	26	0.04	2	0.04	2.44	0.15	1.78
B2	480	474	6	0.01	1	0.03	2.94	0.10	4.57
B3	10	2	8	0.80	1	1.44	1.85	3.74	5.05
ALL	1370	1329	41	0.03	2	0.04	2.94	0.14	5.05

Table 9
Better than the best known solutions achieved

Set	Instance	Value
B1	N4C3W4_O	226
B1	N4C3W4_P	219
B1	N4C3W4_R	214
B1	N4C3W4_T	216

solved on average in 7.45 and 360.69 s, respectively, on a 120 MHz Pentium. This is in comparison with our processing times of 0.07 and 0.06 s, respectively on a 400 MHz Pentium. Taking the relative powers of the two computing media into consideration, it is still possible to say that our algorithms are considerably more computationally efficient, particularly for the T class problems.

Scholl et al. [2] study the well-known heuristics, first fit descending (FFD), best fit descending (BFD), worst fit descending (WFD), as well as the B2F heuristic, which works like FFD until a bin is filled then tries to exchange the smallest item assigned to the bin with two small-sized items not currently assigned such that the residual capacity is decreased. They also introduce another heuristic (FFD-B2F), which combines FFD with B2F. Our MBS′ is comparable in terms of computational requirements to these heuristics, but turns out to be superior in terms of solution quality (see Table 11).

Scholl et al. also present a sophisticated dual tabu algorithm and an exact branch and bound algorithm (BISON). Table 12, where the results for BISON are shown for time limits (TL) equal to 50 and 1000 s, shows that our final algorithm is competitive.

Table 10
Comparison (Pert. MBS′ = Perturbation MBS′)

Algorithm	hits	Inferior	Abs. dev.		Rel. dev.		Time	
			av.	max.	av.	max.	av.	max.
MBS	678	692	1.02	9	1.42	16.67	0.05	8.99
MBS′	1051	319	0.38	9	0.61	14.29	0.02	4.57
Relaxed MBS′	1172	198	0.16	6	0.31	14.29	0.09	5.88
Perturbation MBS′	1176	194	0.22	5	0.38	14.29	0.08	4.57
Sampling MBS′	1198	172	0.14	2	0.24	5.56	0.10	16.63
VNS	1248	122	0.09	2	0.18	5.00	0.09	4.57
Pert. MBS′ + VNS	1329	41	0.03	2	0.04	2.94	0.14	5.05

Table 11
Hits for MBS′ vs. other heuristics

Set	*N*	FFD	BFD	WFD	B2F	FFD-B2F	MBS′
B1	720	547	548	409	545	617	629
B2	480	236	236	192	292	319	381
B3	10	0	0	0	0	0	0

Table 12
Hits for Pert. MBS′ + VNS vs. other algorithms (TL = time limit)

Set	*N*	Dual Tabu	BISON TL = 50	BISON TL = 1000	Pert. MBS′ + VNS
B1	720	666	695	697	694
B2	480	466	472	473	474
B3	10	3	3	3	2

6.7. Robustness test

Since both algorithms used in the final solution scheme, i.e., the perturbation MBS′ and the variable neighbourhood search, involve an element of randomisation, the question of its robustness arises. To answer this question each benchmark problem instance has been solved 10 times, starting from a different random seed each time. In the overwhelming majority of cases (1340 out of 1370), the same solution was achieved every run. For the remaining 30 instances, the difference between the best and the worst number of bins achieved was always equal to 1.

7. Summary and conclusions

A number of new heuristics to solve the bin packing problem have been presented. Some improvements on the MBS heuristic of Gupta and Ho [5] have been made. Moreover, a modified version, called MBS′, has been designed. The main characteristic of this new heuristic is fixing one item in a bin before the MBS one-packing search for this bin is invoked.

Based on MBS′, three new heuristics have been designed: relaxed MBS′, which allows non-zero slack in bins, perturbation MBS′, which uses the one-packing procedure to improve the existing solution by building new packings, and sampling MBS′, which invokes MBS′ several times, randomly changing the order of the items.

Although the MBS′ heuristic and its derivative heuristics have a high computational complexity, extensive computational experiments show empirically that they are very efficient.

An efficient and effective variable neighbourhood search scheme has also been developed and the design of its basic components presented.

Computational experiments show that all these heuristics give very good results in reasonably short processing times. The final algorithm, consisting of perturbation MBS′ followed by variable neighbourhood search, together with a good set of lower bounds, is a remarkably effective and efficient tool for solving the bin packing problem.

The success of the VNS algorithm indicates that the VNS metaheuristic does provide a useful framework for tackling many operational research problems. Due to the flexibility of this framework, it may be possible to design an even more effective algorithm for one-dimensional bin packing using more sophisticated swaps of items and/or other types of moves in shaking and local search.

Future work could explore the possibility of designing more sophisticated MBS-based heuristics that combine creatively several of the ideas developed in this work, such as that of allowing slack, improving existing solutions and sampling. On the other hand, the solution strategies presented here could conceivably be used to solve effectively similar problems, like the simple assembly line balancing.

References

[1] Garey MR, Johnson DS. Computers and intractability: a guide to the theory of NP-completeness. San Francisco: W.H. Freeman, 1979.
[2] Scholl A, Klein R, Jürgens C. BISON: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. Computers & Operations Research 1997;24(7):627–45.
[3] Valério de Carvalho JM. Exact solution of bin-packing problems using column generation and branch-and-bound. Annals of Operations Research 1999;86:629–59.
[4] Gupta JND, Ho JC. A new heuristic algorithm for the one-dimensional bin-packing problem. Production Planning & Control 1999;10(6):598–603.
[5] Fekete SP, Schepers J. New classes of lower bounds for bin packing problems, Lecture Notes in Computer Science, vol. 1412. Berlin: Springer, 1998. p. 257–70.
[6] Hoffmann TR. Assembly line balancing with a precedence matrix. Management Science 1963;9:551–62.
[7] Martello S, Toth P. Knapsack problems. New York: Wiley, 1990.
[8] Kolisch R, Hartmann S. Heuristic algorithms for the resource-constrainted project scheduling problem: classification and computational analysis. In: Węglarz J editor. Project scheduling: recent models, algorithms and applications. Dordrecht: Kluwer, 1999.

[9] Hansen P, Mladenović N. An introduction to variable neighbourhood search. In: S. Voss et al., editors. Metaheuristics, advances and trends in local search paradigms for optimization. Dordrecht: Kluwer, 1999. p. 433–58.
[10] Mladenović N, Hansen P. Variable neighbourhood search. Computers & Operations Research 1997;24 (11):1097–100.
[11] Beasley JE. OR-library: distributing test problems by electronic mail. Journal of the Operational Research Society 1990;41(11):1069–107.

Krzysztof Fleszar is a student at the Faculty of Electronics and Information Technology of Warsaw University of Technology. He spent the 1999/2000 academic year working as visiting researcher at the Department of Systems Engineering of Brunel University. Mr Fleszar's research interests are in scheduling and combinatorial optimisation.

Khalil S. Hindi is a Professor of Systems Engineering at Brunel University, Greater London, England. He is a Fellow of the following British learned societies: the Institution of Electrical Engineers, the British Computer Society, the Institute of Mathematics and its Applications and the Royal Society for the Encouragement of Arts, Manufactures and Commerce. His current research interests are in computer-aided management, planning, operation, scheduling and control of engineering systems; particularly manufacturing systems, electric power systems and gas and water distribution and transmission systems.