Consistent neighborhood search for one-dimensional bin packing and two-dimensional vector packing

Mirsad Buljubašić, Michel Vasquez

Consistent neighborhood search for one-dimensional bin packing and two-dimensional vector packing

Mirsad Buljubašić *, Michel Vasquez

Ecole des Mines d'Alès, LGI2P Laboratory, Nîmes, France

A B S T R A C T

We propose a consistent neighborhood search approach for solving the one-dimensional bin packing problem (BPP). The goal of this local search is to derive a feasible solution with a given number of bins, m, starting from $m = UB -$ 1, where UB is an upper bound obtained by using a variant of the classical First Fit heuristic. To this end, the local search was performed on a partial solution with $m - 2$ bins, i.e. a solution containing a subset of items packed into $m - 2$ bins without capacity violations and a set of non-assigned items, with the objective of minimizing the total weight of non-assigned items and, ultimately, packing all the non-assigned items into two bins. A partial solution was constructed by deleting bins from the last complete solution. Local moves include rearranging the items assigned to a single bin along with non-assigned items, i.e. removing and adding items to the bin. A tabu search was performed with moves featuring a limited number of items to be added/dropped, plus a hill climbing/descent procedure with general (unlimited) add/drop moves, in order to minimize a given objective function. The very same procedure was used for all instances under consideration, with the same initial solution, same para-meters, same order of neighborhood exploration, etc. Promising results were obtained for a wide range of benchmark instances; solutions equal to or better than the best known solutions found by heuristic methods were obtained for all the instances considered, successfully outperforming published results for the particular class of instances hard28, which appears to cause the greatest degree of difficulty for BPP algorithms. The method was also tested on the vector packing problem (VPP) and evaluated for classical two-dimensional VPP (2-DVPP) benchmarks, in all instances yielding optimal or best-known solutions.

Keywords:
Tabu search
Consistent neighborhood
Bin packing
Vector bin packing

1. Introduction

Given a set $I = \{1, 2, \ldots, n\}$ of items with associated weights w_i $(i = 1, \ldots, n)$, the bin packing problem (BPP) consists in finding the minimum number of bins, of capacity C, required to pack all the items without violating any of the capacity constraints. In other words, the goal is to find a partition of items $\{I_1, I_2, \ldots, I_m\}$ such that

$$\sum_{i \in I_j} w_i \leq C, \quad j = 1, \ldots, m$$

and m is minimum. The bin packing problem is known to be NP-hard [10]. It is one of the most extensively studied combinatorial problems and has a wide range of practical applications such as storage allocation, cutting stock, multiprocessor scheduling and the loading of flexible manufacturing systems, to name a few. The Vector Packing problem (VPP) is a generalization of BPP with multiple resources. Item weights w_i^r and bin capacities C_r are defined for each resource $r \in \{1, \ldots, R\}$, and the following constraint must be satisfied:

$$\sum_{i \in I_j} w_i^r \leq C_r, \quad r = 1, \ldots, R, j = 1, \ldots, m.$$

Without loss of generality, we assume that capacities and weights are integer-valued.

This paper will present a new improvement heuristic based on a local search for solving BPP and VPP with two resources (2-DVPP). The method will first be described in detail for a BPP problem, followed by adaptations required to solve the 2-DVPP. The solution is iteratively improved by decreasing the number of bins being utilized. The procedure works as follows. First, the upper bound on the solution value, UB, is obtained by a variant of the First Fit heuristic. Next, an attempt is made to find a feasible solution with $UB - 1$ bins, and this process continues until the lower bound, time limit or maximum number of search iterations

* Corresponding author.
E-mail addresses: mirsad.buljubasic@mines-ales.fr (M. Buljubašić), michel.vasquez@mines-ales.fr (M. Vasquez).

is reached. Apart from the simple lower bound, $\left\lceil \frac{\sum_{i=1}^{n} w_i}{C} \right\rceil$, other lower bounds developed by Fekete and Schepers [7], Martello and Toth [13] (bound L_3) and Alvim et al. [1] are also used.

In order to find a feasible solution with a given number of bins, $m < UB$, a local search is employed. As opposed to the majority of papers published on BPP, the local search explores partial solutions that consist of a set of assigned items without any capacity violation and a set of non-assigned items. The moves rearrange the items assigned to a single bin along with non-assigned items, i.e. items are removed and added to the bin. The objective here is to minimize the total weight of non-assigned items. This local search on partial configurations is called the Consistent Neighborhood Search (since only valid partial packings are considered). It has been proven efficient on several combinatorial optimization problems [22,24]. Our approach will therefore be referred to as CNS_BP (Consistent Neighborhood Search for Bin Packing) in the remainder of the paper.

This search space of partial solutions is explored in two successive phases: (1) a tabu search with limited add/drop moves and (2) a descent with a general add/drop move. This sequence terminates when a complete solution is found or the running time limit or maximum number of iterations is reached. Additionally, the algorithm makes use of a simple reduction procedure that consists in fixing the assignments of all pairs of items that can fill an entire bin. More precisely, once a set of item pairs (i,j) such that $w_i + w_j = C$ is identified, the problem can be reduced by deleting those items (or setting their assignments). This same reduction has been used in most papers on BPP. It is important to mention that the reduction procedure does not have a significant influence on the final results (but can speed up the search) and that no reduction is possible for a large proportion of the instances considered.

This paper is organized as follows. Section 2 will describe relevant work. Then, our approach will be introduced in Section 3. The general framework will be presented first, followed by a description of all its algorithmic components. A number of critical remarks and parameter choices will be discussed in Section 4. Section 5 presents a summary of methodological adaptations to solve the 2-DVPP. The results of extensive computational experiments performed on the available set of instances, for both the BPP and 2-DVPP, will be provided in Section 6, followed by a conclusion.

2. Relevant work

2.1. BPP

There is a large body of literature concerning the one-dimensional bin packing problem. Both exact and heuristic methods have been applied for solving the problem. Martello and Toth [13] proposed a branch-and-bound procedure (MTP). Scholl et al. [17] developed a hybrid method (BISON) that combines a tabu search with a branch-and-bound procedure based on several bounds and a new branching scheme. Schwerin and Wäscher [18] offered a new lower bound for the BPP based on the cutting stock problem, then integrated this new bound into MTP and achieved high-quality results. Valerio de Carvalho [21] proposed an exact algorithm using column generation and branch-and-bound.

Gupta and Ho [11] introduced a minimum bin slack (MBS) constructive heuristic. At each step, a set of items that fits the bin capacity as tightly as possible is identified and packed into the new bin. Fleszar and Hindi [9] developed a hybrid algorithm that combines a modified version of the MBS and the Variable Neighborhood Search. Their hybrid algorithm performed well in computational experiments, by producing the optimal solution for 1329 out of the 1370 instances considered (the first two classes of instances to be discussed in Section 6.1).

Alvim et al. [1] presented a hybrid improvement heuristic (HI_BP) that uses tabu search to move the items between bins. In their algorithm, a complete yet infeasible configuration is to be repaired through a tabu search procedure. Simple “shift and swap” neighborhoods are explored, in addition to balancing/unbalancing the use of bin pairs by solving a Maximum Subset Sum problem. HI_BP performed very well, having obtained the optimal solution for 1582 out of the 1587 instances considered (the first four classes of instances to be discussed in Section 6.1).

In recent years, several competitive heuristics have been presented with results similar to those obtained by HI_BP. Singh and Gupta [19] proposed a compound heuristic (C_BP) which combines a hybrid steady-state grouping genetic algorithm with an improved version of Fleszar and Hindi's Perturbation MBS. Loh et al. [12] developed a weight annealing (WA) procedure, by relying on the concept of weight annealing to expand and accelerate the search by creating distortions in various parts of the search space. The proposed algorithm is simple and easy to implement; moreover, the authors reported high-level performances, exceeding those obtained by HI_BP.

Fleszar and Charalambous [8] offered a modification to the Perturbation-MBS method [9] where a new sufficient average weight (SAW) principle is introduced to control the average weight of items packed in each bin (referred to as Perturbation-SAWMBS). This heuristic outperformed the best state-of-the-art HI_BP, C_BP and WA algorithms. The authors also reported significantly lower quality results for the WA heuristic compared to those given in Loh et al. [12].

To the best of our knowledge, the most recent work in this area, is reported in Quiroz-Castellanos et al. [16]. It involves a grouping genetic algorithm (GGA-CGT) that outperforms all previous algorithms with regard to the number of optimal solutions found, particularly for the most difficult set of instances hard28. The authors propose a new set of grouping genetic operators to promote the transmission of the best genes in the chromosomes. A new reproduction technique that controls the exploration of the search space is also presented, as well as a variant of the First Fit procedure for producing a high-quality initial population.

Brandão and Pedroso [3] devised an exact approach for solving the bin packing and cutting stock problems based on an Arc-Flow Formulation of the problem which is then solved with the commercial Gurobi solver. They were able to optimally solve all standard bin packing instances within a reasonable computation times, including those instances that were not solved to optimality by any heuristic method.

2.2. VPP

With regard to the two-dimensional VPP, Spieksma [20] proposed a branch-and-bound algorithm, while Caprara and Toth [4] reported exact and heuristic approaches as well as a worst-case performance analysis. A heuristic approach using a set-covering formulation was presented by Monaci and Toth [15]. Masson et al. [14] proposed an iterative local search (ILS) algorithm for solving the Machine Reassignment Problem and VPP with two resources; they reported the best results for the classical VPP benchmark instances of Spieksma [20] and Caprara and Toth [4].

3. Proposed heuristic

This section will describe our improvement heuristic. The main part of the improvement procedure is illustrated in Fig. 1, while

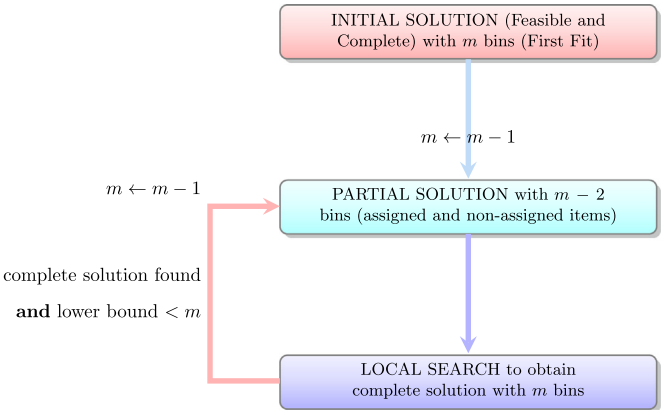

Fig. 1. General improvement procedure.

the pseudo-code of the full procedure is given in Algorithm 1.

The algorithm starts by applying a simple reduction procedure and by constructing an initial (feasible and complete) solution by applying the First Fit heuristic on a randomly sorted set of items. The random sort is used to avoid solutions with many small or big non-assigned items, which could make the search more difficult or slower (this is the case, for example, it items are sorted in decreasing order). This initial solution, containing UB bins, is then improved by a local search-based procedure, which represents the core element of our proposal. More precisely, an attempt is made to find a complete solution with $m = UB - 1$ bins by applying a local search on a partial solution, and this process is repeated until a lower bound, time limit or maximum number of iterations is reached (precisely defined later).

Algorithm 1. CNS_BP

remove item pairs (i,j) such that $w_i + w_j = C$
compute lower bound LB
randomly shuffle the set of items
$S \leftarrow$ complete solution obtained by First Fit
$m \leftarrow$ number of bins in S
While $m > LB$ **and** time limit not exceeded **do**
  $m \leftarrow m - 1$
  build partial solution P with $m-2$ bins ▷ delete 3 bins from S
  $S' \leftarrow CNS(P)$ ▷ try to find complete solution with m bins
  If solution S' not complete **then** TERMINATE
$S \leftarrow S'$
end while
  return S ▷ return the last complete solution

The remainder of this section will describe a procedure aimed at finding a feasible solution with a given number of bins, m. The basic idea here is to consider a partial solution with $m-2$ bins and then transform it into a complete feasible solution with m bins. The partial solution contains a set of items assigned to $m-2$ bins, without any capacity violation, and a set of non-assigned items. The local search, by rearranging the items, then tries to obtain a configuration such that the non-assigned items can be packed into two bins, thus producing a complete feasible solution with m bins. This procedure is illustrated in Fig. 2.

It should be noted that a complete solution cannot be obtained if more than two "big" items (with weight greater than or equal to half of the bin capacity) are not assigned (i.e., non-assigned items cannot be packed into two bins). Therefore, the maximum number of non-assigned big items is limited to two throughout the procedure. When it is possible to pack the non-assigned items into two bins, the complete solution is obtained by simply adding the two new bins to the current set of bins.

The partial solution with $m-2$ bins is obtained by deleting three bins from the last complete solution obtained with $m+1$ bins i.e. by removing all the items from these bins and by adding them to the set of non-assigned items. Bins to be deleted are selected in the following way:

- the last two bins from a complete solution,
- the last bin (excluding the last two) such that total number of non-assigned "big" items does not exceed two.

The capacity of the bins cannot be violated at any time during the procedure.

For the sake of simplicity, let us assume that the non-assigned items are packed into the special bin with unlimited capacity, called *trash can* and denoted by TC. Let $B = \{b_1, b_2, \ldots, b_{m-2}\}$ be the set of currently utilized bins, $I_B \subseteq I$ the set of items assigned to the bins in B and I_b the set of items currently packed into bin $b \in B$. Similarly, let I_{TC} denote the set of items not currently assigned to any bin. The total weight and cardinality of a set of items S will be denoted by $w(S)$ and $|S|$ respectively. The total weight and number of items currently assigned to bin $b \in B \cup TC$ will be denoted by $w(b) = w(I_b)$ and $|b| = |I_b|$.

3.1. Local search

The Local Search procedure is applied to reach a complete solution with m bins, starting from a partial one with $m-2$ bins. Several neighborhoods are explored during the search, which consists of two procedures executed in succession until a stopping criterion is reached. These two procedures are: (a) tabu search procedure and (b) hill climbing/descent procedure. All moves consist of swapping the items between a bin in B and the trash can TC.

Formally speaking, the local search moves include:

1. $Swap(p, q)$ – swap p items from a bin $b \in B$ with q items from TC.
2. $Pack(b)$ – optimally rearrange the items between bin $b \in B$ and trash can TC, such that the remaining capacity in b is minimized, that is, the set of items assigned to bin b fits the bin capacity as tightly as possible. *Pack* is a generalization of a *Swap* move with p and q both being unlimited.

Only moves not resulting in any capacity violation are considered during this search.

Algorithm 2. $CNS(sol)$.

input: partial solution sol
while time or iterations limit not exceeded **and** complete solution not found **do**
  $sol \leftarrow TabuSearch(sol)$
  $sol \leftarrow Descent(sol)$
end while
return sol

Swap moves are used only in the tabu search procedure, while the descent procedure exclusively makes use of pack moves. The pseudocode of the procedure is given in Algorithm 2. The two main parts, *TabuSearch()* and *Descent()*, will be explained later in the corresponding sections.

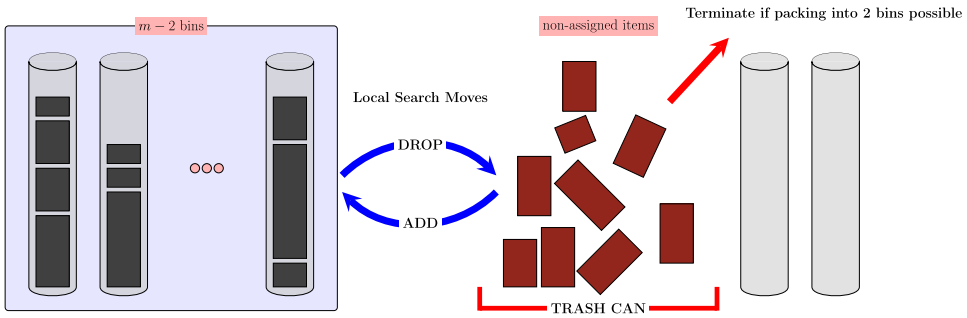

Fig. 2. Consistent neighborhood search.

But before, we will discuss the objective function, search neighborhoods and search termination conditions.

The goal of the local search procedure is to optimize the following lexicographic objective function:

1. Minimize the total weight of non-assigned items (minimize use of the trash can): min $w(TC)$.
2. maximize the number of items in the trash can: max $|TC|$.

The first objective is quite natural, while the second is introduced in order to yield items with lower weights in the trash can, as this could: (1) increase the chance of terminating the search; and/or (2) enable a wider exploration of the search space. Formally, the following function is to be minimized:

$$n \times w(TC) - |TC|.$$

Objective function of the solution s will be denoted by $obj(s)$.

The maximum number of items from the same bin that can be rearranged in a single *Swap* move is limited to three. More precisely, $Swap(p, q)$ moves with

$$(p, q) \in PQ = \{(0, 1), (1, 1), (2, 1), (1, 2), (2, 2), (1, 3), (3, 1), (2, 3), (3, 2)\}$$

are considered. $Swap(0, 1)$ corresponds to *shift* move, which consists of shifting (or adding) the item from the trash can to bin $b \in B$. Note that the higher complexity of these $Swap(p, q)$ moves, with respect to the classical shift and swap moves used in the literature, is compensated by the fact that no moves between pairs of bins in B are performed.

Generating an optimal packing for a set of items, as originally proposed in Gupta and Ho [11], is a common procedure introduced in several papers [9,8]. The *Pack* move is the same as the "load unbalancing" used in Alvim et al. [1]. The packing problem is equivalent to the Maximum Subset Sum (MSS) problem and can be solved exactly through dynamic programming or by enumeration. In our case, the packing procedure is only being used for a small subset of items, i.e. the set of items belonging to a single bin $b \in B$ or trash can TC. For a set of items S, let $pack\_set(S)$ denote the solution to the MSS problem, which is a subset $P \subseteq S$ of maximum total weight such that $w(S) \leq C$. The enumeration procedure was used here and its pseudocode is given in Algorithm 3. Clearly, the complexity of the enumeration procedure is $O(2^l)$, where l is a number of items considered. Our experiments showed that using a dynamic programming procedure (of complexity $O(l \times C)$) instead of enumeration does not produce better solutions for the set of instances considered, while improvement in the total running time of the algorithm, if any, is negligible.

As mentioned above, no more than two "big" items (with weights greater than or equal to $C/2$) can be assigned to the trash can during the entire problem solving procedure. This is easily achieved by forbidding all the moves that result in three or more big items in the trash can. This is omitted in the algorithms presented below (pseudocodes) for the sake of simplicity.

Algorithm 3. $pack\_set(S, P)$.

input: set of items S, current packing P (initialized to empty set)
output: best packing P^* (static variable, initialized to empty set)
If $S = \emptyset$ **then**
  If $w(P) > w(P^*)$ **or** $w(P) = w(P^*)$ **and** $|P| < |P^*|$ **then** $P^* \leftarrow P$
else
  $i \leftarrow$ first item in S
  $S \leftarrow S \setminus \{i\}$
  If $w(P) + w_i \leq C$ **then** $pack\_set(S, P \cup \{i\})$
  $pack\_set(S, P)$
end if

During the search, each time the total weight in the trash can is less than or equal to $2C$, an attempt is made to pack all items from the trash can in two bins. Obviously, this is possible if and only if

$$w(pack\_set(I_{TC})) \geq w(TC) - C.$$

If packing into two bins is indeed feasible, then a complete solution with the target number of bins has been found and the local search procedure terminates. The local search procedure also terminates when the total number of solutions with $w(TC) \leq 2C$ obtained during the search exceeds a given number. It seems reasonable to terminate the search after failing too many times to pack the non-assigned items into two bins, and this limit is set to 100,000 times for all the instances considered. In addition, further exploration of the search space does not appear to be worthwhile if no solution with $w(TC) \leq 2C$ can be obtained in a reasonable time. Therefore, the search terminates if no solution with $w(TC) \leq 2C$ has been found during the first ten algorithm loops (tabu + descent).

3.1.1. Tabu search

The main component of the improvement procedure is a tabu search that includes *Swap* moves between trash can and bins in B. The whole procedure is given in Algorithm 4. At each iteration, all feasible and non-tabu $Swap(p, q)$ moves between the trash can and each bin are evaluated and the best one with regard to the defined objective is performed. Should two or more moves with the same best objective value exist, then a random choice is made. Note that the best move is carried out even if it does not improve the solution. Each time the total weight in the trash can is less than or equal to $2C$, an attempt is made to terminate the search by packing the non-assigned items into two bins.

This process is repeated until no feasible and non-tabu move exists, or until the time limit *timeLimitTabu* or the maximum number of consecutive moves without improvement (*maxNmbIters*) is reached. All the results reported in Section 6 were obtained using *timeLimitTabu* = 1 *sec* and *maxNmbIters* = $|B| \times |I|$. The tabu search procedure returns either the best solution found if the

initial solution is improved or the last solution obtained (see the last 5 lines in Algorithm 4).

Whenever an item with weight w is placed into bin b via a swap move, all swap moves that include an item from b with weight w become tabu for a specific number of iterations. Only removing the items from the bin (i.e. placing them into the trash can) can be considered tabu. Thus, moving an item from the trash can to a bin is never tabu. The number of iterations for which moving an item of weight w from bin b to the trash can is tabu depends on the number of swap moves performed that place an item of weight w into b, $freq(b, w)$. More precisely, the given move is tabu for $freq(b, w)/2$ iterations. All frequencies $freq(b, w)$ are reset to zero when the current best objective value is improved in the tabu search. Implementation of the tabu list management is straightforward and procedures *resetTabu*(), *isTabu*() and *updateTabu*() (given on page 14) are invoked in the main algorithm (Algorithm 4) to reset, check and update the tabu status respectively.

Given that our objective function is lexicographic, minimizing the total weight of non-assigned items has a higher priority than maximizing the number of non-assigned items. Nevertheless, this choice can lead the search to configurations in which the first objective is quite well optimized, but with a very small number of non-assigned items, which might make the search termination and exploration of the search space more difficult. We therefore decided to use two different variants of the tabu search procedure, namely:

- tabu search consisting of all defined $Swap(p, q)$ moves,
- tabu search consisting of a subset of swap moves that do not decrease the second objective, that is, $(p, q) \in \{(1, 1), (2, 1), (2, 2), (3, 1), (3, 2)\}$.

The two variants differ only in the set of allowed swap moves and are applied one after the other. Thus, the whole tabu search procedure consists in sequentially applying these two variants. The second variant significantly improved the results obtained on the hard 28 dataset (around 5 new optimal solutions on average).

Algorithm 4. *TabuSearch*().

input: initial solution *initSol*
$sol \leftarrow initSol$
$bestSol \leftarrow initSol$
resetTabu()
while $iter < maxNmbIters$ **and** running time $< timeLimitTabu$ **do**
  $S^* \leftarrow \emptyset$, $T^* \leftarrow \emptyset$, $b^* \leftarrow -1$
  $\min\Delta \leftarrow n \times w(I)$
  for each $b \in B$ **do**
    for each $S \subseteq I_b$ **do**
      If $isTabu(b, S) = false$ **then** ▷ only non-tabu moves
        for each $T \subseteq I_{TC}$, $(|S|, |T|) \in PQ$. **do** ▷ only allowed pairs
          if $w(b) + w(T) - w(S) \leq C$ **then** ▷ check capacity
            $\Delta \leftarrow n \times (w(S) - w(T)) - (|S| - |T|)$
            if $\Delta \leq \min\Delta$ **then**
              $S^* \leftarrow S$, $T^* \leftarrow T$, $b^* \leftarrow b$, $\min\Delta \leftarrow \Delta$
            end if
          end if
        end for
      end if
    end for
  end for
  if $T^* = \emptyset$ **then** TERMINATE ▷ terminate when no allowed move exists swap S^* and T^* in sol ▷ perform $Swap(|S^*|, |T^*|)$ move
  if $obj(sol) < obj(bestSol)$ **then** ▷ update best sol.
    $bestSol \leftarrow sol$
    resetTabu()
  end if
if $w(TC) \leq 2C$ **and** $w(pack\_set(I_{TC})) \geq w(TC) - C$ **then** TERMINATE
end if
updateTabu(b^*, S^*)
end while
if $bestSol \neq initSol$ **then return** *bestSol*
else return *sol*

resetTabu()
$iter \leftarrow 0$
for each pair $(b, i) \in B \times I$ **do** $freq[b][w_i] \leftarrow 0$, $tabu[b][w_i] \leftarrow -1$

isTabu(b, S)
input: bin $b \in B$, set of items $S \subseteq I_b$ to drop from b
for each item $i \in S$ **do if** $tabu[b][w_i] \geq iter$ **return** true
return false

updateTabu(b, S)
input: bin $b \in B$, set of items $S \subseteq I_{TC}$ to add to bin b
  $iter \leftarrow iter + 1$
for each item $i \in S$
  $freq[b][w_i] \leftarrow freq[b][w_i] + 1$
  $tabu[b][w_i] \leftarrow iter + freq[b][w_i]/2$

3.1.2. Descent procedure

The second part of the local search explores the solution space by applying *Pack* moves. The latter are exclusively used in the descent (hill climbing) procedure due to their greater complexity. As in the tabu search, these moves are performed only between the trash can and bins in B. The $Pack(b)$ move is executed for each bin $b \in B$ until no improvement in the objective can be achieved. Formally, $Pack(b)$ consists in assigning a set of items $pack\_set(I_b \cup I_{TC})$ to bin b and set $(I_b \cup I_{TC}) \setminus pack\_set(I_b \cup I_{TC})$ to the trash can. The bins in B are considered in random order. It is clear that the objective cannot increase during the procedure due to the nature of the pack move. As in the tabu search, a feasible packing of non-assigned items into two bins is attempted after each *Pack* move such that $w(TC) \leq 2C$. The running time of the descent procedure is significantly less than that of the tabu search, which is understandable given that all performed moves must provide an improvement.

A complete $Pack(b)$ move is performed only when the total number of items considered $|I_b \cup I_{TC}|$ does not exceed 20. Otherwise, a limited $Pack(b)$ move on a random subset of items containing no more than ten items from b and no more than ten items from TC is performed. Thus, the complexity of the *Pack* move never exceeds 2^{20}. The descent procedure is described in Algorithm 5.

Algorithm 5. *Descent*().

repeat
  randomly sort the set of bins B
  foreach $b \in B$ **do**
    $Pack(b)$ ▷ perform *Pack* move between b and TC
    if $w(TC) \leq 2C$ **and** $w(pack\_set(I_{TC})) \geq w(TC) - C$ **then**
    TERMINATE
  end if
end for
Until no objective improvement made

4. Discussion and parameters

Unlike many published algorithms, only one procedure has been implemented to build an initial solution. We decided to use First Fit because of its simplicity, while randomly sorting the set of items is used to avoid search configurations with many items of similar size (too many small or too many big items for example) in the trash can. The trash can could contain too many small items if, for example, the First Fit Decrease heuristic is used. Furthermore, solutions obtained by using only the First Fit heuristic on a given set of items (without random shuffling) depend on the order of items in the benchmark data files; they may be in decreasing order for example, or even in an order such that the First Fit heuristic would produce an optimal solution for each instance.

Many experiments including choices of neighborhoods, termination conditions, etc. have been carried out. For instance, a partial solution with $m-3$ bins can be employed, and the termination condition could be the packing of the non-assigned items into three bins. This modification can produce improved average results or computation times for certain instances, though our experiments showed that no overall improvement can be achieved. On the other hand, the procedure could be simplified by adopting a partial solution with $m-1$ bins (or m bins) and using $w(TC) \leq C$ (or $w(TC) = 0$) as the termination criteria. The first approach with $m-1$ bins would produce similar results for most of the datasets considered, while the second one yields significantly lower quality solutions on all datasets.

Since the tabu search procedure is the main part of the algorithm, one can expect that the maximum number of items to be rearranged in a single $Swap(p, q)$ move, i.e. the upper bounds for p and q, can significantly influence solution quality. As will be shown later, the results obtained by allowing no more than two items from the same bin to be swapped, rather than three, are only slightly worse. However, allowing just $Swap(0, 1)$ and $Swap(1, 1)$ moves, drastically reduces solution quality. This outcome is to be expected since capacity violation is prohibited and most moves quickly become infeasible. Our algorithm uses only a few parameters. Its total running time was limited to 60 s for all instances. Opting for a much smaller limit would produce the same results for most instances, but the 60 s limit is preferred due to the difficulty of the instances belonging to the hard28 and gau_1 classes. The tabu search procedure was limited to a duration of *timeLimitTabu*, which was set to one second in all reported experiments, and a maximum number of iterations (moves) without improvement, *maxNmbIters*, set to $|B| \times |I|$. The complete *Pack* move was only performed if the total number of items considered for rearrangement did not exceed 20 (which is not really a limitation for any of the instances considered). Otherwise moves were limited to a subset of 20 items. This limitation was introduced to avoid excessive enumeration times, but similar values (15–25 for example) would produce the same quality of results. Furthermore, if a dynamic programming procedure is used to get optimal packings, no limitation on the number of items is required. The tabu tenure in the tabu search procedure is proportional to the frequency of the move. The list of all the parameters is given in Table 1.

Optimizing the running time was not our primary goal, so we did not devote much effort to accelerating the algorithm or finding solutions more quickly for certain instances through different modifications (e.g., exploring fewer neighborhoods). No distinction whatsoever was made between the instances. Nevertheless, we sought to obtain an algorithm with reasonable running times, and we believe this goal has been achieved.

Table 1
Algorithm parameters: the first three can be considered as algorithmic choices, while the other six are experimentally set to given values.

Parameter	Chosen value
Bins in partial solution	$m-2$
Termination condition	$w(TC) \leq 2C$
Max. number of items from the same bin in *Swap* move	3
Local search running time limit	60 s
Max. number of items in *Pack*() move	20
Max. number of iterations without improvement in tabu search (*max NmbIters*)	$\|B\| \times \|I\|$
TabuSearch() running time limit (*timeLimitTabu*)	1 s
Max. number of attempts made to pack non-assigned items	100,000
Tabu tenure	frequency/2

5. Applying the method to 2-DVPP

The generalization of the presented method to 2-DVPP is straightforward. The main modification involves the objective function. Let $w^1(TC)$ and $w^2(TC)$ be the total weight of non-assigned items on resources 1 and 2, respectively. The first objective here is to minimize the larger of the two values, $w^1(TC)/C_1$ and $w^2(TC)/C_2$. The second objective, as for BPP, is to maximize the number of items in the trash can. Formally, the following function is to be minimized:

$$n \times C_1 \times C_2 \times \max\left(\frac{w^1(TC)}{C_1}, \frac{w^2(TC)}{C_2}\right) - |TC|.$$

As for the BPP problem, the upper bound *UB* is obtained by applying the First Fit heuristic to a randomly sorted set of items. The reduction procedure is analogous to the one used in BPP: a set of item pairs (i,j) s.t. $w_i^1 + w_j^1 = C_1$ and $w_i^2 + w_j^2 = C_2$ is identified, and the problem is reduced by assigning these items. In the case of 2-DVPP, we consider the weight of item j to be greater than or equal to half of the bin capacity if $w_j^1 \geq \frac{C_1}{2}$ and $w_j^2 \geq \frac{C_2}{2}$.

Only a simple lower bound,

$$\left\lceil \max\left(\frac{\sum_{i=1}^{n} w_i^1}{C_1}, \frac{\sum_{i=1}^{n} w_i^2}{C_2}\right)\right\rceil,$$

is used for 2-DVPP. All other algorithmic features, such as the chosen parameters, remain the same. An analogous adaptation could be made for VPP with more than two resources; however, experiments were not conducted since only a few relevant experimental results are reported in the literature and most algorithms have only been tested on 2-DVPP benchmarks.

6. Computational results

This section will report the results of extensive computational experiments performed with our method on a broad set of test problems. Our algorithm was implemented in C++ and compiled using the gcc 4.7.2 compiler in Ubuntu 14.04. All tests were run on a computer with an Intel Core i7-3770 CPU 3.40 GHz processor. The computation times are reported in seconds. Unless otherwise specified, the reported values always correspond to the total running time of the algorithm i.e. from the calculation of the lower bound and construction of the initial solution to the search termination.

6.1. BPP

A common set of one-dimensional bin-packing instances was used to test our algorithm. This set consists of five classes of

instances: (1) a class developed by Falkenauer [6] made of two sets, uniform and triplets (denoted respectively by U and T in the tables of results) each with 80 instances; (2) a class developed by [17] consisting of three sets set_1, set_2, and set_3 with 720, 480 and 10 instances respectively; (3) a class of instances developed by Scholl et al. [18] made of two sets, was_1 and was_2, each with 100 instances; (4) a class of instances developed by Schwerin and Wäscher [23], called gau_1, containing 17 problem instances; (5) the hard28 class, consisting of 28 difficult problems instances used, e.g. in [2]. All instances can be downloaded from the Web page of the EURO Special Interest Group on Cutting and Packing [5]. Optimal solutions for all instances are known.

Optimal solutions for the first three classes (1570 instances in all) were obtained using several heuristics, including HI_BP [1] and GGA-CGT [16]. HI_BP optimally solves 12 of the 17 instances in gau_1, while other recent heuristics yielded more optimal results (e.g. 15 by C_BP [19] and 16 by Perturbation-SAWMBS [8] and GGA-CGT). The only instance from this class that could not be solved to optimality by any heuristic algorithm was "TEST0014".

Fleszar and Charalambous reported that their Perturbation-SAWMBS method could not solve to optimality more instances in the hard28 dataset than the First Fit Decrease procedure (5 out of the 28), even after drastically increasing the maximum number of iterations in their algorithm. The same applies to the HI_BP algorithm, as reported in Quiroz-Castellanos et al. [16]. In fact, most heuristics proposed for the bin packing problem, including the best performers, cannot solve to optimality more than 5 instances from this class. Nevertheless, GGA-CGT finds optimal solutions to 16 instances. The authors also reported that more instances can be solved by increasing the population size (up to 22 instances when the population is increased from 500 to 10,000,000).

The exact method based on the Arc-Flow Formulation, as presented in Brandão and Pedroso [3], can solve all instances to optimality within a reasonable computing time, including all instances from the hard28 dataset. The solver can be downloaded at http://vpsolver.dcc.fc.up.pt/ and detailed results obtained using the Gurobi solver to solve their Arc-Flow model are given at http://www.dcc.fc.up.pt/fdabrandao/research/vpsolver/results/.

Average running times of this exact algorithm for each class of instances are listed in column *time* in Table 2. The computer used was a 2 × Quad-Core Intel Xeon at 2.66 GHz, Mac OS X 10.8.0, 16 GB of memory, while the Gurobi 5.0.0 solver (single thread) was used to solve the model. It can be noted that the computation times in their experiments were much longer for the gau_1 dataset (up to a few thousand seconds) when compared to other datasets because the average number of items per bin in this dataset is greater than in other datasets. Generally, an exact algorithm runs until optimality is proven, although it might find the optimal (or feasible) solution very early, and can therefore be, for example, easily transformed into a heuristic approach by stopping it after a given time limit. However, terminating the exact algorithm based on the Arc-Flow Formulation before optimality is proven is not of much value because the lower bound obtained through the linear relaxation is optimal in the vast majority of cases (>99%), and the algorithm terminates almost as soon as an optimal solution if found. Column *IPtimeToBest* of Table 2 shows the average running time (excluding the time for calculating a bound) to reach the optimal solution. Table 3 reports the total number of optimal solutions found by the exact approach with different computation time limits (excluding the time spent for calculating the bound).

Table 2
Results of the exact approach based on the Arc-Flow formulation.

Class	inst	time (s)	IPtimeToBest (s)
U	80	0.34	0.24
T	80	0.91	0.71
set_1	720	0.15	0.10
set_2	480	43.4	39.7
set_3	10	12.1	7.53
was_1	100	0.67	0.52
was_2	100	0.57	0.40
gau_1	17	1641	1485
hard28	28	29.69	27.0

Table 3
Number of optimally solved instances after limiting the running time of the exact approach based on the Arc-Flow formulation.

Turning the exact approach into a heuristic					
Time limit (s)	60	120	300	600	1000
Optimal solutions	1520	1558	1598	1607	1611

To investigate the effectiveness of CNS_BP, we compared these results with those obtained by the best heuristic approaches reported in the literature, Perturbation-SAWMBS and GGA–CGT.

The running time of CNS_BP is highly influenced by several important factors, such as the lower bound used to terminate the search and the limits on running time and number of iterations. Using a more complicated bound could terminate the search earlier but may also consume significant CPU time. Choosing appropriate limits on running time and number of iterations largely depends on the set of instances to be solved. For example, if the hard28 dataset was excluded from consideration, which is the case in many published papers on BPP, then the average total time would decrease substantially, as would the required limits on running time and iterations. In the reported results, there is no distinction made in this regard. Raising the limit on running time or number of iterations can increase the number of optimal solutions but can also drastically increase the average running time since the total running time is reported even if no improvement is achieved. We also report the running times required to obtain the best solution values i.e. excluding the time spent exploring the search space after the last complete solution has been found. All the results reported in this section were obtained with a running time limit of 60 s and no more than 100,000 moves resulting in $w(TC) \leq 2C$, unless otherwise specified. The lower bound used for each instance was the best of the following four bounds: $L_1 = \left\lceil \frac{\sum_{i=1}^{n} w_i}{C} \right\rceil$, $L^{(p)}$ proposed in Fekete and Schepers [7] with $p=10$, L_3 bound from Martello and Toth [13] and L_ϑ proposed in Alvim et al. [1].

Like most previous work on BPP, a single execution of the algorithm was performed on each instance, with the initial seed for the random number generation set to 1. The results are reported in Table 4. For each class of instances, the number of instances solved to optimality with each approach, as well as the average computation time per instance, are reported. For our algorithm, the average time to reach the best solution (*toBest*) per instance for each class is also reported. Given that the algorithms were executed on different machines, we calculated appropriate scaling factors to compare their run times. For this purpose, we used the CPU speed estimations provided the in SPEC standard benchmark (https://www.spec.org/cpu2006/results/cint2006.html). On this basis, the CPU speeds of the processors used to run Perturbation-SAWMBS, GGA-CGT, and CNS_BP are 18.30, 12.30 and 53.80 respectively. If we choose the second one as a reference, the scaling factors will be 1.48, 1 and 4.37 respectively. By multiplying the CPU time of each algorithm by its corresponding scaling factor calculated above, we obtain a Unified Computational Time (UCT) [25]. In Table 4, computational time expressed in UCT is given for P.-

Table 4
Results with a seed set equal to 1. A comparison is drawn with the best state-of-the-art methods.

Results for seed = 1										
Class	Inst	P.-SAWMBS			GGA-CGT		CNS_BP			
		Opt	Time	scTime	Opt	Time	Opt	Time	scTime	toBest
U	80	79	0.00	0.00	80	0.23	80	0.068	0.299	0.058
T	80	80	0.00	0.00	80	0.41	80	0.018	0.078	0.018
set_1	720	720	0.01	0.015	720	0.35	720	0.073	0.319	0.002
set_2	480	480	0.00	0.00	480	0.12	480	0.033	0.144	0.033
set_3	10	10	0.16	0.24	10	1.99	10	0.004	0.017	0.004
was_1	100	100	0.00	0.00	100	0.00	100	0.000	0.002	0.000
was_2	100	100	0.01	0.015	100	1.07	100	0.001	0.004	0.001
gau_1	17	16	0.04	0.06	16	0.27	17	2.679	11.70	1.822
hard28	28	5	0.24	0.36	16	2.40	25	7.209	31.50	2.007
Total	1615	1590			1602		1612			
		Intel core2 Q8200 2.33 GHz			Core2 Duo CE6300 1.86 GHz		Intel i7-3770 3.40 GHz			

SAWMBS and CNS_BP, while it is not necessary for the GGA-CGT algorithm since the corresponding factor is equal to 1.

A more detailed and more relevant (in our opinion) result for CNS_BP would be the average obtained by running the algorithm with 100 different seeds (1–100) and with different time limits. These results are reported in Table 5. As in the previous table, the number of instances solved to optimality and average running time per instance are indicated for each class and each of five time limits (60, 30, 10, 5 and 2 s). In terms of number of optimal solutions found, our algorithm outperforms all published heuristic algorithms on the last two datasets and obtains the same (optimal) solutions on all other datasets. Perturbation-SAWMBS is faster in terms of running time, while GGA-CGT and CNS_BP are comparable in this regard.

The number of solved instances from the hard28 dataset varies from 24 to 27 for the various seeds as reported in Table 8. The only unsolved instance over 100 runs is "BPP_13". Nevertheless, optimal solution for this instance can be found by increasing the time limit or running the algorithm with more seeds.

The total number of instances solved to optimality (out of 1615) varies between 1607 and 1613 when running the algorithm with 100 seeds. More details on the distribution of the results are given in Table 6. The results obtained with certain algorithmic simplifications are reported in Table 7. More precisely, we report the results obtained after prohibiting more than two items from the same bin to be rearranged in a single *Swap* move during the tabu search and the results obtained without the *Descent*() procedure. This table reports the average results over 100 runs. One can notice that these two simplifications do not significantly impact the quality of the results.

Table 6
Number of runs (out of 100) producing a given number of optimal solutions.

Results distribution							
Optimally solved instances	1607	1608	1609	1610	1611	1612	1613
Number of runs	2	3	11	26	32	17	9

Table 7
Results with simplifications, average results for 100 seeds; results when allowing a maximum of two items from the same bin to be swapped in the tabu search procedure and results without applying the Descent procedure.

Class	Inst	*Swap*($p \leq 2, q \leq 2$)		no *Descent*()	
		Opt	Time	Opt	Time
U	80	80.0	0.063	80.0	0.069
T	80	80.0	0.013	80.0	0.016
set_1	720	719.26	0.059	719.37	0.074
set_2	480	479.94	0.065	479.03	0.125
set_3	10	10.0	0.104	10.0	0.001
was_1	100	100.0	0.025	100.0	0.000
was_2	100	100.0	0.022	99.91	0.004
gau_1	17	16.05	3.958	16.92	2.186
hard28	28	24.60	7.573	24.73	7.379
Total	1615	1609.85		1609.96	

6.2. 2-DVPP

The 2-DVPP instances used to evaluate the performance of CNS_BP were extracted from Spieksma [20] and Caprara and Toth [4]. A total of 10 different classes of instances were used. Each class is composed of 40 instances, broken down into 10 instances of four

Table 5
Average results of CNS_BP for 100 runs and for different time limits.

Class	60 s		30 s		10 s		5 s		2 s	
	Opt	Time	Opt	Time	Opt	Time	Opt	Time	Opt	Time
U	80.0	0.069	80.0	0.069	80.0	0.069	80.0	0.069	80.0	0.069
T	80.0	0.016	80.0	0.016	80.0	0.016	80.0	0.016	80.0	0.016
set_1	719.31	0.074	719.31	0.069	719.31	0.069	719.31	0.069	719.31	0.069
set_2	479.83	0.043	479.82	0.042	479.57	0.038	479.35	0.035	478.68	0.031
set_3	10.0	0.001	10.0	0.001	10.0	0.001	10.0	0.001	10.0	0.001
was_1	100.0	0.000	100.0	0.000	100.0	0.000	100.0	0.000	100.0	0.000
was_2	100.0	0.001	100.0	0.001	100.0	0.001	100.0	0.001	100.0	0.001
gau_1	16.77	2.600	16.53	2.039	16.21	1.323	16.08	0.874	16.5	0.388
hard28	24.81	7.385	24.46	4.751	23.86	2.655	23.13	1.745	21.45	0.945
Total	1610.72		1610.12		1608.95		1607.87		1605.49	

Table 8
Detailed results for the hard28 dataset. The number of runs resulting in an optimal solution (#opt) is reported for each instance, as are the average running time (*time*) and average time spent to obtain best solutions (*toBest*). 100 runs were conducted for each instance. It should be noted that the average running time over all instances is largely influenced by the running time for the most difficult instances and for instances where the lower bound is different from the optimal value (BPP_119 for example).

Instance	LB	OPT	#opt	Time	toBest
BPP_14	61	62	100	10.180	0.000
BPP_832	60	60	100	0.553	0.553
BPP_40	59	59	31	50.279	9.343
BPP_360	62	62	100	0.049	0.049
BPP_645	58	58	100	2.125	2.125
BPP_742	64	64	100	0.066	0.066
BPP_766	62	62	91	9.386	7.266
BPP_60	63	63	23	2.039	0.231
BPP_13	67	67	0	7.660	0.000
BPP_195	64	64	100	0.048	0.048
BPP_709	67	67	36	48.464	10.412
BPP_785	68	68	100	0.235	0.235
BPP_47	71	71	100	0.110	0.110
BPP_181	72	72	100	1.337	1.337
BPP_359	75	76	100	1.532	0.000
BPP_485	71	71	100	0.645	0.645
BPP_640	74	74	100	0.046	0.046
BPP_716	76	77	100	1.106	0.000
BPP_119	76	77	100	59.374	0.000
BPP_144	73	73	100	0.854	0.854
BPP_561	72	72	100	0.013	0.013
BPP_781	71	71	100	0.019	0.019
BPP_900	75	75	100	2.510	2.510
BPP_175	83	84	100	3.840	0.001
BPP_178	80	80	100	0.241	0.241
BPP_419	80	80	100	3.974	3.974
BPP_531	83	83	100	0.068	0.068
BPP_814	81	81	100	0.030	0.030

different sizes. The Class 10 instances are generated by cutting the bin resources into triplets of items, such that all bins are full in optimal solutions. For this class, therefore, the optimal solutions in most cases are known from the instance generation process, and are not a result of the bin packing algorithms. Classes 2, 3, 4, 5 and 8 are known to be easily solvable by simple greedy heuristics [15], hence our experiments focus on the remaining classes, a total of 200 instances.

The previous best results have been obtained by the iterated local search (MS-ILS) heuristic reported in Masson et al. [14]. Thus, we compare our results to theirs. It should be noted that 330 out of 400 instances could still be solved by the exact approach proposed by [3]; 60 out of the 70 unsolved instances belong to classes 4 and 5 and can be easily solved optimally with non-exact approaches. Consequently, 10 instances belonging to class 9 and containing 200 items are the only ones with unknown optima. Our method obtained optimal solutions for all instances with known optima (390 out of 400) and the best known values for the remaining 10 instances.

The results reported in Table 9 have been aggregated by problem class, i.e. for each class the cumulative number of bins for the 10 instances is reported. A total of 50 runs with different seeds were performed for each instance. From left to right, the columns in Table 9 report the problem size, problem class, simple lower bound (defined in Section 5), optimal value (if known), the best known upper bound (obtained by MS-ILS), average running time per instance for MS-ILS, average and best number of bins generated with our heuristic, and the average CPU time per instance and average CPU time required to obtain the best solutions. All the best known upper bounds and optimal solutions from previous algorithms were found. The running time limit was set to 10 s, and no more than two items from the same bin could be rearranged in a single *Swap* move throughout the tabu search procedure. The MS-ILS results reported in Masson et al. [14] were obtained with a time limit of 300 s on an Opteron 2.4 GHz with 4 GB of RAM memory running Linux OpenSuse 11.1. According to the CPU speed estimations provided in the SPEC standard benchmark (https://www.spec.org/cpu2006/results/cint2006.html), this machine is around five times slower than the one we used in our experiments (more details on the Opteron 2.4 GHz are required for a more accurate value). Therefore, the MS-ILS running times reported in Masson et al. [14] have been divided by five.

Table 9
2-DVPP Results. Improvements over previous solutions found by heuristics are shown in bold.

2-DVPP results - 50 runs									
N	Class	LB	OPT	MS-ILS		CNS_BP			
				Best	Time	Avg	Best	Time	toBest
25	1	69	69	69	2.54	69.0	69	0.000	0.000
25	6	99	101	101	4.26	101.0	101	2.000	0.000
25	7	95	96	96	3.72	96.0	96	1.000	0.000
25	9	63	73	73	4.06	73.0	73	10.000	0.000
24	10	80	80	80	2.22	80.0	80	0.000	0.000
50	1	135	135	135	14.5	135.0	135	0.000	0.000
50	6	213	215	215	13.7	215.0	215	2.000	0.000
50	7	196	197	197	17.6	197.0	197	1.000	0.000
50	9	135	145	145	39.8	145.0	145	10.000	0.000
51	10	170	170	170	13.8	170.0	170	0.000	0.000
100	1	255	255	257	58.9	255.0	**255**	0.034	0.034
100	6	405	410	410	60.0	410.0	410	5.001	0.001
100	7	398	402	402	57.8	402.0	402	4.004	0.004
100	9	257	267	267	60.0	267.0	267	10.000	0.000
99	10	330	330	330	46.4	330.0	330	0.007	0.007
200	1	503	503	503	60.0	503.0	503	0.011	0.011
200	6	803	811	811	60.0	811.0	811	8.002	0.002
200	7	799	801	802	60.0	801.0	**801**	2.000	0.000
200	9	503	—	513	60.0	513.0	513	10.011	0.011
201	10	670	670	678	60.0	670.1	**670**	0.913	0.784

7. Conclusion

In this paper, we propose a new local search-based algorithm for the Bin Packing Problem. The main feature of this algorithm is to proceed by partial configurations of the search space. Items are assigned to $m - 2$ bins, while satisfying the capacity constraint, or are not assigned at all. This algorithm seeks to derive a set of non-assigned items that can be packed into two bins to obtain a complete feasible solution with m bins. Subsequently, the computation is divided into two repeated steps:

- the tabu search applies only low cardinality swap moves between non-assigned items and bins. The objective function is aimed at minimizing the sum of weights of non-assigned items while maximizing the number of non-assigned items;
- the descent procedure generates locally optimal packings between non-assigned items and each bin using the same objective function. This procedure performs generalized swap moves between a bin and non-assigned items.

Whenever the total weight of non-assigned items is less than or equal to twice the capacity of a bin, an attempt is made to pack all non-assigned items into two bins by using the packing algorithm of the Descent procedure. This algorithm uses few parameters and outperforms all previous heuristic approaches in terms of the number of instances solved to optimality. Let us note in particular

that it obtains better solutions than other heuristics for the hard28 and gau_1 datasets. Considering the simplicity of the overall method, it was quite easy to adapt it to the Vector Packing Problem and solve the available benchmarks with a significantly better performance than other published approaches.

These results might come as a surprise because the algorithm only focuses on the configuration of non-assigned items, which may be viewed as a major restriction. But, integrating bin characteristics into the objective function (*like* $\sum_{b \in B} (C - w(b))^2$ *for instance*), increases *CPU* time without providing better results.

Acknowledgements

The authors would like to thank the two anonymous reviewers and the area editor for their constructive comments that improve the content as well as the presentation of the paper.

References

[1] Alvim ACF, Ribeiro CC, Glover F, Aloise DJ. A hybrid improvement heuristic for the one-dimensional bin packing problem. J Heurist 2004;10:205–29.
[2] Belov G, Scheithauer G. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. Eur J Oper Res 2006;171(1):85–106.
[3] Brandão, F, Pedroso JP. Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression. Technical Report DCC-2013-08, Faculdade de Ciências da Universidade do Porto, Portugal; 2013.
[4] Caprara A, Toth P. Lower bounds and algorithms for the 2-dimensional vector packing problem. Discrete Appl Math 2001;111(3):231–62.
[5] ESICUP, 2013. Euro Especial Interest Group on Cutting and Packing. One Dimensional Cutting and Packing Data Sets. http://paginas.fe.up.pt/esicup/tiki-list_file_gallery.php?galleryId=1.
[6] Falkenauer E. A hybrid grouping genetic algorithm for bin packing. J Heurist 1996;2:5–30.
[7] Fekete SP, Schepers J. New classes of fast lower bounds for bin packing problems. Math Program 2001;91:11–31.
[8] Fleszar K, Charalambous C. Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem. Eur J Oper Res 2011;210:176–84.
[9] Fleszar K, Hindi KS. New heuristics for one-dimensional bin-packing. Comput Oper Res 2002;29:821–39.
[10] Garey MR, Johnson DS. Computers and intractability: a guide to the theory of np-completeness. New York, USA: W.H. Freeman; 1979.
[11] Gupta JND, Ho JC. A new heuristic algorithm for the one-dimensional bin packing problem. Prod Plann Control 1999;10:598–603.
[12] Loh KH, Golden B, Wasil E. Solving the one-dimensional bin packing problem with a weight annealing heuristic. Comput Oper Res 2008;35(7):2283–91.
[13] Martello S, Toth P. Knapsack problems: algorithms and computer implementations.Chichester, England: Wiley; 1990.
[14] Masson R, Vidal T, Michallet J, Penna PHV, Petrucci V, Subramanian A, et al. An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems. Expert Syst Appl 2013;40(13):5266–75.
[15] Monaci M, Toth P. A set-covering-based heuristic approach for bin-packing problems. INFORMS J Comput 2006;18(1):71–85.
[16] Quiroz-Castellanos M, Cruz-Reyes L, Torres-Jiménez J, Gomez Santillán C, Huacuja HJF, Alvim ACF. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. Comput Oper Res 2015;55:52–64.
[17] Scholl A, Klein R, Jurgens C. Bison: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. Comput Oper Res 1997;24(7):627–45.
[18] Schwerin P, Wäscher G. A new lower bound for the bin-packing problem and its integration into mtp and bison. Pesqui Op 1999;19:111–29.
[19] Singh A, Gupta AK. Two heuristics for the one-dimensional bin-packing problem. OR Spectr 2007;29(4):765–81.
[20] Spieksma FCR. A branch-and-bound algorithm for the two-dimensional vector packing problem. Comput Oper Res 1994;21(1):19–25.
[21] Valerio de Carvalho JM. Exact solution of bin-packing problems using column generation and branch-and-bound. Ann Oper Res 1999;86:629–59.
[22] Vasquez M, Dupont A, Habet D. Consistency checking within local search applied to the frequency assignment with polarization problem. RAIRO – Oper Res 2003;37(4):311–23.
[23] Wäscher G, Gau T. Heuristics for the integer one-dimensional cutting stock problem: a computational study. Oper-Res-Spektrum 1996;18(3):131–44.
[24] Zufferey N, Vasquez M. A generalized consistent neighborhood search for satellite range scheduling problems. RAIRO – Oper Res 2015;49(1):99–121.
[25] Perboli G, Pezzella F, Tadei R. EVE-OPT: a hybrid algorithm for the capacitated vehicle routing problem. Math Methods Oper Res 2008;68(2):361–82.