Computers & Operations Research 36 (2009) 2877 – 2884

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/cor

Heuristics for the variable sized bin-packing problem

Mohamed Haouari[a,b], Mehdi Serairi[b,*]

[a]Department of Industrial and Systems Engineering, Faculty of Engineering, Ozyegin University, Istanbul, Turkey
[b]ROI—Combinatorial Optimization Research Group, Ecole Polytechnique de Tunisie, BP 743, 2078, La Marsa, Tunisia

ARTICLE INFO

Available online 6 January 2009

Keywords:
Bin-packing problem
Heuristics
Genetic algorithm

ABSTRACT

We investigate the one-dimensional variable-sized bin-packing problem. This problem requires packing a set of items into a minimum-cost set of bins of unequal sizes and costs. Six optimization-based heuristics for this problem are presented and compared. We analyze their empirical performance on a large set of randomly generated test instances with up to 2000 items and seven bin types. The first contribution of this paper is to provide evidence that a set covering heuristic proves to be highly effective and capable of delivering very-high quality solutions within short CPU times. In addition, we found that a simple subset-sum problem-based heuristic consistently outperforms heuristics from the literature while requiring extremely short CPU times.

1. Introduction

Bin-packing problems constitute a challenging class of combinatorial optimization problems having a wealth of pertinence to a wide range of applied areas including packing, cutting, and scheduling. In this paper, we investigate heuristic algorithms for the *variable sized bin-packing problem* (*VSBPP*) which is a generalization of the well-known one-dimensional bin-packing problem (*1BPP*). The *VSBPP* is formally defined as follows. We are given m different bin types, where each bin type i ($i=1,\ldots,m$) is characterized by a *capacity* W_i and a *fixed cost* c_i. The *VSBPP* requires packing a set J of n items, where each item j ($1\leqslant j\leqslant n$) has a *weight* w_j, into a minimum-cost set of bins while ensuring that the total weight of all items loaded into a bin does not exceed the bin's capacity. Practical applications arise in packing, transportation planning, and cutting as well.

So far, several authors have investigated approximate solution strategies for the *VSBPP* and its variants. Approximation algorithms giving absolute and/or asymptotic worst-case bounds have been proposed and analyzed. Friesen and Langston [1] presented three approximation algorithms, their asymptotic worst-case performance bounds are, respectively, 2, $\frac{3}{2}$, and $\frac{4}{3}$, respectively. Murgolo [2] developed a polynomial-time approximation scheme. Coffman et al. [3] claimed that a modified *first fit decreasing* (FFD) algorithm gives an optimal solution when the item weights are divisible and the bin capacities are multiples of all the item weights. However, Kang and Park [4] presented a counter-example and showed that this latter result was not correct. Chu and La [5] described four approximation algorithms having absolute worst-case performances 2, 2, 3 and $2+\ln 2$, respectively. In addition, Kang and Park [4] proposed two variants of the well-known FFD and *best-fit decreasing* algorithms, respectively, and analyzed their asymptotic worst-case performance in three cases. They show that the algorithms give a solution which is less than $\frac{3}{2}z^*+1$ for the general case and less than $\frac{11}{9}z^*+4\frac{11}{9}$ for the case when only the bin capacities are divisible (where z^* refers to the value of an optimal solution). Interestingly, they prove that when both the item weights and the bin capacities are divisible the algorithms give optimal solution. At this point it is worth mentioning that all the above quoted papers, but Kang and Park's paper, deal with the special case where $W_i=c_i\ \forall i=1,\ldots,m$. In addition to these approximate solution strategies, exact methods have been proposed by Monaci [6] Belov and Scheithauer [7] Alves and Valério de Carvalho [8], and Haouari and Serairi [9]. However, since the *VSBPP* is $\mathcal{NP}$-hard these exact methods fail to solve large-scale instances within a reasonable CPU time. In this paper, we propose and analyze the empirical performance of six heuristics for the *VSBPP*. A common feature to all the proposed procedures is that they embed (exact) optimization algorithms. We present the results of extensive computational results that provide strong evidence of the efficacy of some proposed heuristics. In particular, we report the derivation of very near-optimal solutions for *VSBPP* instances with up to 2000 items and seven different bin types within very short CPU times.

The remainder of this paper is organized as follows. In Section 2, we describe four constructive heuristics that require iteratively solving a subset-sum problem (SSP). Next, we investigate, in Section 3, a set covering-based heuristic. In Section 4, we describe an optimization-based genetic algorithm (GA). In Section 5, we report

* Corresponding author.
E-mail address: mehdi.serairi@gmail.com (M. Serairi).

0305-0548/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2008.12.016

the results of extensive computational experiments. Finally, some concluding remarks are provided in Section 6.

In the sequel, we shall assume w.l.o.g. that $W_1 < W_2 < \cdots < W_m$ and $c_1 < c_2 < \cdots < c_m$.

2. Constructive heuristics

2.1. A subset-sum based-heuristic and its variants

In this section, we describe heuristics that iteratively build a feasible *VSBPP* solution by exactly solving a sequence of SSP. The first heuristic, hereafter denoted by *SSP* 1, might be viewed as a generalization of the subset-sum heuristic for 1*BPP* [10]. Indeed, *SSP* 1 is an iterative constructive heuristic that selects at each iteration a new empty bin and fills it with a subset of unpacked items. The bins are selected and filled in the following way. Define $\bar{J}$ as the set of unpacked items at iteration k (obviously, at the first iteration we set $\bar{J}=J$). First, we compute for each bin type i ($i=1,\ldots,m$) the maximum load z_i by solving the following SSP:

$$z_i = \text{Maximize} \quad \sum_{j\in\bar{J}} w_j x_j \tag{1}$$

背包

subject to

$$\sum_{j\in\bar{J}} w_j x_j \leqslant W_i, \tag{2}$$

$$x_j \in \{0,1\}, \quad \forall j \in \bar{J}. \tag{3}$$

It is well-known that although being an $\mathcal{NP}$-hard problem, the SSP can be efficiently solved in pseudo-polynomial time (see [11]). Let $x^i \in \{0,1\}^{|\bar{J}|}$ denote the corresponding optimal solution and define $S_i = \{j \in \bar{J} : x^i_j = 1\}$ the set of selected items. Second, we compute for each bin type i ($i=1,\ldots,m$) the ratio of the fixed cost c_i to the maximal load z_i and we select the bin type i^* having the smallest ratio (that is, $i^* = \arg\min_{1\leqslant i\leqslant m}\{c_i/z_i\}$). Third, we drop from $\bar{J}$ the items of S_{i^*} and we pack them into a bin of type i^*. This process is reiterated until all items are packed. A pseudo-code description of *SSP* 1 is the following.

Heuristic SSP1.

Step 0—Initialization: $\bar{J}=J$
Step 1—For $i=1$ to m
  Solve (1)–(3) to compute z_i and S_i
Step 2—Set $i^* = \arg\min_{1\leqslant i\leqslant m}\{c_i/z_i\}$
Step 3—Load the items of S_{i^*} into a bin of type i^*. Set $\bar{J}=\bar{J}\backslash S_{i*}$
Step 4—If $\bar{J}=\varnothing$ then Stop, else go to Step 1.

A variant of this heuristic, hereafter referred to by *SSP* 2, requires that at each iteration of Step 1, only those bins that can include the largest unpacked items are considered for possible filling. Hence, at each iteration, the SSP is only solved for the bin types for which the inequality $W_i \geqslant \max_{j\in\bar{J}} w_j$ holds. Furthermore, we have considered a variant of *SSP* 2, where we enforce at each iteration the largest unpacked item to be included in the set of items that are packed into the selected bin. In the sequel, we refer to this latter variant by *SSP* 3. It is noteworthy that Caprara and Pferschy [12] proposed and analyzed similar modified subset-sum heuristics in the context of 1*BPP*.

2.2. A one-dimensional bin-packing-based heuristic

The fourth heuristic, hereafter denoted by *SSP* 4, is a constructive heuristic that aims at delivering m feasible solutions that are obtained by successively considering the different bin types. This heuristic is described as follows. At iteration i ($i=1,\ldots,m$), we initialize $k=i$, the set of unpacked items $\bar{J}=J$, and we determine the set J_k of items that can be packed into a bin of type k. That is $J_k=\{j\in\bar{J} : w_j\leqslant W_k\}$. Then, we solve an 1*BPP* defined on the subset J_k and identical bins having a capacity W_k. In our implementation, the 1*BPP* is approximately solved using the subset-sum heuristic [10] where bins are filled in turn, loading into each bin a subset of unpacked items of J_k of maximum weight not exceeding the bin capacity W_k. Clearly, if we find that the total weight of a filled bin can be loaded into a smaller bin, then the capacity of this bin will be accordingly decreased to the smallest possible value. Next, we update the set of unpacked items $\bar{J}=\bar{J}\backslash J_k$. Obviously, if $\bar{J}=\emptyset$ then a feasible packing π_i has been obtained. Otherwise, we set $k=k+1$ and the process is reiterated. Finally, we select the least-cost packing among $\pi_1,\ldots,\pi_m$. A pseudo-code of *SSP* 4 is described below.

Heuristic SSP4:

For $i=1$ to m do
Begin
  0. Set $\bar{J}=J$, $k=i$
  1. Set $J_k=\{j\in\bar{J} : w_j\leqslant W_k\}$
  2. Solve an 1BPP defined on J_k
  3. Assign to each filled bin the smallest feasible capacity
  4. Set $\bar{J}=\bar{J}\backslash J_k$
  5. Test: if $\bar{J}\neq\emptyset$ then Set $k=k+1$ and go to 1
  6. Store the obtained solution π_i
End (For)
Output the best obtained solution

3. A set covering-based heuristic

Prior to describing the set covering-based heuristic (*SC*), we present a set partitioning formulation of *VSBPP*. For each bin type i ($i=1,\ldots,m$), define Π_i as the set of feasible packing. With each packing $k\in\Pi_i$ are associated a binary incidence vector $a_{ik}=(a_{ik1},\ldots,a_{ikn})$, such that $\sum_{j=1}^{n} a_{ikj}w_j \leqslant W_i$, and a binary decision variable x_{ik}. This variable has a cost c_i and takes value 1 if packing $k\in\Pi_i$ is included in the solution, and 0 otherwise. A formulation of *VSBPP* is

$$\text{Minimize} \quad \sum_{i=1}^{m}\sum_{k\in\Pi_i} c_i x_{ik} \tag{4}$$

subject to

$$\sum_{i=1}^{m}\sum_{k\in\Pi_i} a_{ikj}x_{ik} = 1, \quad j=1,\ldots,n \tag{5}$$

$$x_{ik}\in\{0,1\}, \quad i=1,\ldots,m, \ \ k\in\Pi_i. \tag{6}$$

The objective function (4) is to minimize the total cost of the selected bins. Constraint (5) imposes that each item is assigned to exactly one feasible packing. Finally, Constraints (6) require that the decision variables are binary-valued. It is easy to check that an equivalent optimal solution could be obtained by solving a set covering formulation that is derived by requiring that each item is assigned to *at least* one feasible packing. However, it is well-known that the linear programming relaxation of the set partitioning problem is often very difficult to solve due to high degeneracy and numerical instability. Therefore, for the sake of computational convenience, the set covering formulation is preferred. It is worth mentioning that Model (4)–(6) is similar to the formulation that has been proposed by Alves and Valério de Carvalho [8] for the exact solution of the linear-cost *VSBPP*. However, the major drawback of the set partitioning/covering formulation is the huge, though finite, number of columns. In order to overcome this difficulty, we propose a two-phase heuristic. First, we generate a restricted subset of "attractive" feasible columns

$\bar{\Pi}_i \subset \Pi_i$ $(i = 1, \ldots, m)$. Next, we derive an *approximate* solution by solving the following restriction of Model (4)–(6):

$$\text{Minimize} \quad \sum_{i=1}^{m} \sum_{k \in \bar{\Pi}_i} c_i x_{ik} \tag{7}$$

subject to

$$\sum_{i=1}^{m} \sum_{k \in \bar{\Pi}_i} a_{ikj} x_{ik} \geqslant 1, \quad j = 1, \ldots, n, \tag{8}$$

$$x_{ik} \in \{0, 1\}, \quad i = 1, \ldots, m, \ k \in \bar{\Pi}_i. \tag{9}$$

This main difference between this two-phase set covering-based heuristic and the classical exact column generation approach, that is often used for solving problems having a huge number of variables, lies in the fact that the columns are generated prior to solving the set covering model and *not* dynamically during the solution process. At this point, it is worth mentioning that similar set covering/partitioning-based heuristics have been previously proposed for solving several hard combinatorial optimization problems. A non-exhaustive list includes the papers by Cullen et al. [13] and Kelly and Xu [14] in the context of vehicle routing, Marsten and Shepardson [15], Wedelin [16], and Caprara et al. [17] for solving airline and railway crew scheduling, and Monaci and Toth [18] in the context of two-dimensional bin packing.

A crucial issue in the implementation of an effective set covering-based heuristic is the careful design of the procedure that is used to generate the restricted set of columns. Ideally, this set should not be too large (so that the resulting set covering problem could be efficiently solved), and also should include high-quality columns that would permit the derivation of a high-quality near-optimal solution. To that aim, we have implemented a probabilistic heuristic that delivers several feasible *VSBPP* solutions. This heuristic might be viewed as a multi-pass variation of the aforementioned bin-packing-based heuristic (*SSP* 4). Indeed, *SSP* 4 is modified in the following way. Instead of solving the 1*BPP* s using the subset-sum-based heuristic, we use a *randomized* version of the well-known *FFD* heuristic. This latter heuristic requires sorting the items in nonincreasing weight. Then at each iteration, one item is randomly selected out of the two heaviest unpacked items and is either assigned to the first bin where it fits, or if this item does not fit into any open bin, into a new empty bin. This process is reiterated until all items are packed. Hence, we obtain a randomized version of *SSP* 4 that is repeated *iter* times. After experimentation, we set the parameter *iter* to 20. In doing so, we obtain a set of $m \times iter$ feasible solutions. In addition, we append to this set all the solutions that are generated by *SSP* 1, *SSP* 2, *SSP* 3, and *SSP* 4, respectively. However, the derived column set may include several useless identical columns that should be discarded prior to solving the set covering problem. In order to reduce the computational burden of this preprocessing step, we determine for each column a_{ik} the three following labels: (i) α_{ik}: index of the lightest item, (ii) β_{ik}: index of the heaviest item, and (iii) γ_{ik}: cumulated weight of the items. Hence, if two columns have identical labels, then one of them is discarded (though they might actually be *different*).

Since, the set covering problem is known to be $\mathcal{NP}$-hard then its exact solution might require an excessive computing time. Instead, a near-optimal solution can be efficiently derived using a heuristic algorithm. In our implementation, and for the sake of simplicity, we used a general-purpose MIP solver to derive an approximate solution that is delivered within a preset maximum CPU time $T_{\max}$ (in our tests, we set $T_{\max} = 30$ s). Clearly, one could reasonably expect that a better efficacy would be achieved if a tailored sophisticated heuristic is used for solving the set covering model (see for example, [19–22]). However, this latter strategy would require a substantial coding effort.

4. A genetic algorithm

GAs are widely acknowledged as powerful tools for providing high-quality solutions to a wide variety of challenging combinatorial optimization problems. In this section, we describe a GA for solving the *VSBPP*. A distinctive feature of our GA is that it uses as a representation of a *VSBPP* solution a permutation of the n items. Moreover, it requires solving a shortest path problem for computing the corresponding fitness. Now, we provide a detailed description of the components of the GA.

4.1. Solution encoding

Several permutation-based GAs have been proposed so far for solving bin-packing problems. In particular, this coding scheme has been implemented for solving several variants of the two-dimensional bin-packing problem by Jakobs [23], Hopper and Turton [24], and Zhang et al. [25], respectively. Moreover, Falkenauer [26] and Lima and Yakawa [27] investigated the efficacy of permutation-based GAs for 1*BPP*.

In the sequel, an ordered list (i.e. permutation) $\sigma = (\sigma(1), \ldots, \sigma(n))$ of the n items is used to represent a chromosome (i.e. *VSBPP* solution). Given a chromosome σ, a corresponding *VSBPP* solution is computed using the following procedure which is a variant of the so-called *next-fit* algorithm. First, we tentatively assume that we are given a predefined ordered list of q bins $\omega = (\omega(1), \ldots, \omega(q))$. Starting from the first item $\sigma(1)$ and the first bin $\omega(1)$, the items are loaded in turn into the current bin as long as they fit. In case where the current item does not fit, the current bin is closed and the next bin is initialized. The procedure yields a feasible solution when all items are loaded. Now, we show how for a given item permutation σ an associated *minimum-cost VSBPP* solution could be derived (hence, we determine a corresponding optimal bin ordering). To that aim, we define an acyclic digraph $G = (V, A)$ as follows. The set of nodes is $V = J \cup \{n+1\}$ (that is, a node is associated with each item in addition to a dummy node $n + 1 \equiv \sigma(j + 1)$. The arc set A is constructed as follows:

- There is an arc $(\sigma(j), \sigma(j+1))$ for $j = 1, \ldots, n$. The cost of this arc is set equal to the cost of the smallest bin that fits item $\sigma(j)$.
- There is an arc $(\sigma(j), \sigma(k))$ (with $1 \leqslant j < k \leqslant n+1$) if the items $\sigma(j), \ldots, \sigma(k-1)$ can be loaded into a single bin. The cost of this arc is set equal to the cost of the smallest bin that fits all these items together.

Fact 1. A dipath $\mathscr{P} = (\sigma(1), \sigma(j_1), \ldots, \sigma(j_{q-1}), \sigma(n+1))$ in G from node $\sigma(1)$ to node $\sigma(n+1)$ with a total cost $c(\mathscr{P})$ corresponds to a feasible *VSBPP* solution using q bins and having a total cost $c(\mathscr{P})$. In this solution, items $\sigma(1), \ldots, \sigma(j_1 - 1)$ are loaded in a first bin, items $\sigma(j_{p-1}), \ldots, \sigma(j_p - 1)$ $(p = 2, \ldots, q-1)$ are loaded in a p th bin, and items $\sigma(j_{q-1}), \ldots, \sigma(n)$ are loaded in a q th bin.

A straightforward consequence is the following.

Corollary 1. *Given a chromosome σ, an associated minimum-cost VSBPP solution can be computed in $O(|A|)$-time by solving a shortest path problem in G from node $\sigma(1)$ to node $\sigma(n+1)$. The cost of this path corresponds to the fitness of σ.*

Example 1. Consider the following instance with $n = 6$ and $m = 2$. The weights are $w_1 = 2$, $w_2 = 4$, $w_3 = 5$, $w_4 = 7$, $w_5 = 9$, and $w_6 = 10$. The bins' characteristics are $(W_1, c_1) = (12, 3)$ and $(W_2, c_2) = (18, 5)$. Assume that $\sigma(j) = j$ $(j = 1, \ldots, 6)$. The corresponding graph is depicted in Fig. 1. The shortest path is $\mathscr{P} = (1, 5, 6, 7)$. This solution has a cost

染色体表示一个排序，每个染色体用最优路径得到当前的最优背包解
然后排序是利用遗传算法去求解

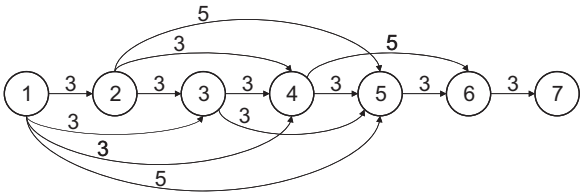

Fig. 1. Graph of Example 1.

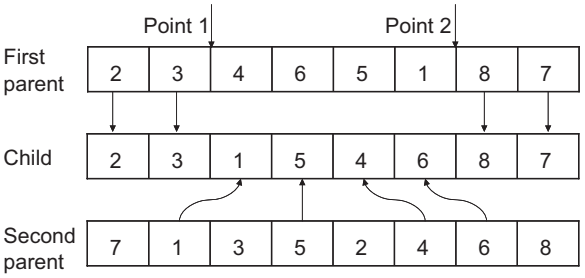

Fig. 2. Two-point crossover.

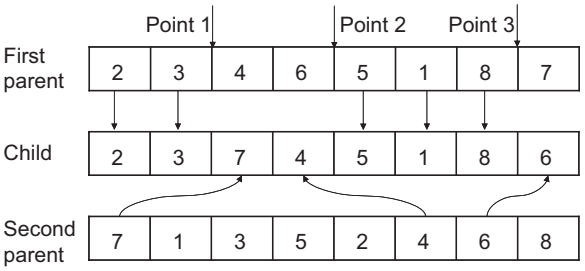

Fig. 3. Three-point crossover.

of 11 and corresponds to loading items 1, 2, 3, and 4 into a bin of Type 2, and items 5 and 6 are loaded into two distinct bins of Type 1, respectively.

4.2. Crossover operators

We have investigated the performance of the following crossover operators.

- *Two-point crossover*: A pair of crossing points is randomly selected along the length of the first parent chromosome. The items outside the selected two points are copied into the offspring and the remaining items are copied from the second parent in the order of their appearance (see Fig. 2).
- *Three-point crossover*: Three crossing points are randomly selected along the length of the first parent. The chromosome is then divided into four distinct sections. The first and third sub-sections are copied into the same places of the offspring and the remaining empty locations of the offspring are filled with items from the second parent according to their order of appearance. No item duplication is permitted (see Fig. 3).
- *Similar job one-point order crossover*: This crossover was introduced by Ruiz et al. [28] for solving a permutation flowshop problem. A crossing point is randomly selected along the length of the first parent, we produce an offspring by copying the identical items at the same position in the parents into the corresponding positions of it. Then missing items before the crossing point are inherited from the first parent. Lastly the missing elements are copied in the relative order of the second parent. It is worth noting that if no identical items are at the same position in the parents, the crossover operator will behave like the one-point crossover (see Fig. 4).

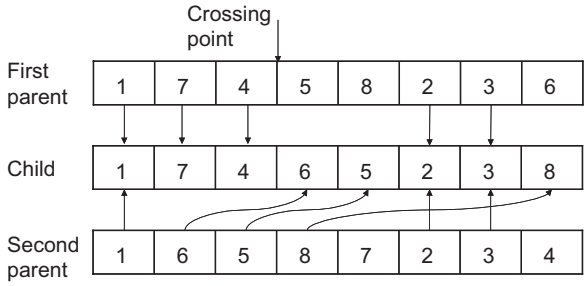

Fig. 4. Similar job one-point crossover.

After performing extensive computational experiments, we found that a good performance is achieved through randomly using the three above-described crossover operators. More precisely, the two-point crossover, the three-point crossover, and the similar job one-point order crossover are invoked with probabilities 0.4, 0.3 and 0.3, respectively.

4.3. Mutation operator

In order to get high-quality solutions, we have hybridized the GA with a powerful local search method acting as a mutation operator. The basic idea is the following. Consider a chromosome σ to be mutated and assume that the associated solution π uses q bins. We select among these loaded bins a subset of $\hat{q}$ bins having a total cost C and including an item subset $\hat{J}$. We invoke *SSP* 3 for solving a restricted *VSBPP* instance that is defined on subset $\hat{J}$. In case where an improved solution is derived (i.e. having a cost strictly less than C), then the $\hat{q}$ selected bins are replaced by the newly generated solution. This search procedure is iterated until no improvement is achieved. Now, we provide two implementation details.

Selection of the bin subset: The $\hat{q}$ bins are selected in the following way. First, we compute for each loaded bin the ratio of the bin cost to the bin actual load. Then, we iteratively select the bin having the largest ratio. This process is stopped when the cardinality of the corresponding selected subset $\hat{J}$ exceeds a preset upper bound κ. In our implementation, we found that good results are obtained with $\kappa = 15$.

Replacement of the mutated chromosome: Assume that the mutation operator produced an improved solution π' that uses q' bins and has a total cost $c(\pi')$. For convenience, each bin p ($p = 1, \ldots, q'$) is represented by a string B_p of the items that are included in that bin (the ordering of these items is arbitrarily selected). A chromosome σ' that is associated to π' is obtained by concatenating, in an arbitrary order, strings $B_1, \ldots, B_{q'}$, respectively. Next, we compute the fitness of σ' by solving a shortest path problem. It is easily realized that the fitness of this chromosome is less than or equal to $c(\pi')$.

4.4. Parameters setting

In our implementation, we set the GA parameters as follows:

- Population size = 200.
- Crossover probability = 0.9.
- Mutation probability = 0.9.

- Maximum number of generations $= 10n$.
- Maximum number of consecutive nonimproving generations before stopping $= 50$.

The initial population consists of the chromosomes corresponding to a mix of all solutions produced by the heuristics *SSP* 1, *SSP* 2, *SSP* 3 and *SSP* 4, and also randomly generated permutations.

5. Computational results

We have coded the six proposed heuristics in C and implemented on a Pentium IV 3.2 GHz Personal Computer with 2 GB RAM. The set covering problem were solved using CPLEX 9.1. We have assessed their performance on two different randomly generated problem sets. In the sequel, we provide a detailed description of these sets and we report the results of our computational experiments.

5.1. Performance on linear-cost benchmark instances

First, we assessed the performance of the proposed heuristics on a set of linear-cost instances (that is, satisfying $W_i = c_i$ $\forall i = 1, \ldots, m$), hereafter denoted by Set A, that were generated as described in Monaci [6] who proposed a branch-and-price algorithm. These instances are generated as follows: The number of bin types is $m = 3$, the number of items $n \in \{25, 50, 100, 200, 500\}$. The weights are drawn randomly from the discrete uniform distribution on [1,100]. The capacities are 100, 120, and 150, respectively. For each value of n, ten instances were generated.

The results are displayed in Table 1. For each heuristic, we provide:

- *Gap_Opt*: Average percentage deviation (over 10 instances) with respect to the value of the *optimal* solution (this value is computed using the exact branch-and-bound algorithm that is described in [9]).
- *M_Gap_Opt*: Maximal value of *Gap* (over 10 instances).
- *Opt*: Number of times it delivers an optimal solution.
- *Time*: Average CPU time in seconds.

We see from Table 1 that *GA* outperforms all the other heuristics. Indeed, GA delivers an optimal solution for 46 instances (out of 50) and exhibits a percentage gap of 0.02%. Moreover, it requires an average CPU time of 0.19 s. On the other hand, *SC* is the second best heuristic since it exhibits a percentage gap of 0.21% but requires longer CPU times for solving large instances. Finally, we observe that all the four constructive heuristics exhibit a quite similar performance and deliver a percentage gap ranging from 1.26% (for *SSP* 3) to 1.58% (for *BFD*), while being extremely fast.

Table 1
Performance of the heuristics of linear-cost instances generated as described in [6].

	n	Gap_Opt	M_Gap_Opt	Opt	Time
FFD	25	3.98	7.41	0	0.00
	50	1.91	3.70	0	0.00
	100	1.07	1.96	0	0.00
	200	0.63	1.11	0	0.00
	500	0.17	0.43	0	0.00
	AVG	1.55	7.41		0.00
BFD	25	4.21	7.41	0	0.00
	50	1.91	3.70	0	0.00
	100	1.07	1.96	0	0.00
	200	0.52	1.11	1	0.00
	500	0.17	0.43	0	0.00
	AVG	1.58	7.41		0.00
SSP 1	25	2.88	6.72	0	0.00
	50	2.28	4.12	1	0.00
	100	1.00	3.13	3	0.00
	200	0.48	1.68	2	0.00
	500	0.24	0.36	1	0.02
	AVG	1.38	6.72		0.00
SSP 2	25	2.88	6.72	0	0.00
	50	2.28	4.12	1	0.00
	100	1.00	3.13	3	0.00
	200	0.48	1.68	2	0.00
	500	0.24	0.36	1	0.00
	AVG	1.38	6.72		0.00
SSP 3	25	2.50	6.11	0	0.00
	50	2.20	4.12	0	0.00
	100	0.81	1.81	2	0.00
	200	0.58	0.82	1	0.00
	500	0.24	0.36	1	0.02
	AVG	1.26	6.11		0.00
SSP 4	25	3.16	5.19	0	0.00
	50	1.87	4.78	1	0.00
	100	1.25	3.13	0	0.00
	200	0.56	1.98	1	0.00
	500	0.12	0.32	2	0.03
	AVG	1.39	5.19		0.01
SC	25	0.54	1.46	4	0.08
	50	0.25	0.52	4	0.10
	100	0.10	0.22	5	0.39
	200	0.06	0.10	4	5.88
	500	0.10	0.25	2	18.43
	AVG	0.21	1.46		4.98
GEN	25	0.00	0.00	10	0.10
	50	0.07	0.37	8	0.11
	100	0.02	0.18	9	0.11
	200	0.01	0.09	9	0.22
	500	0.00	0.00	10	0.39
	AVG	0.02	0.37		0.19

5.2. Performance on large-scale nonlinear-cost instances

This second set includes *VSBPP* instances that were generated as follows:

- The number of bin types is $m = 7$. The capacities are 70, 100, 130, 160, 190, 220 and 250.
- The number of items $n \in \{100, 200, 500, 1000, 2000\}$.
- The weights are drawn randomly from the discrete uniform distribution on [1,250].
- Three different cost functions were generated: (i) linear-cost function (Class B1): $c_i = W_i$, $i = 1, \ldots, m$, (ii) concave-cost function (Class B2): $c_i = \lceil 10\sqrt{W_i} \rceil$, $i = 1, \ldots, m$, and (iii) convex-cost function (Class B3): $c_i = \lceil 0.1 W_i^{3/2} \rceil$, $i = 1, \ldots, m$.

We combined these problem characteristics to obtain 15 different problem classes. For each combination, 10 instances were generated.

Table 2 displays a summary of a computational comparison of the six heuristics. For each heuristic, we provide:

- *Gap*: Average percentage deviation (over 10 instances) with respect to a *lower bound* that is derived by solving a network flow-based relaxation that is described in Haouari and Serairi [9].
- *M_Gap*: Maximal value of *Gap* (over 10 instances).
- *Best*: Number of times it yields the best solution among the presented algorithms.
- *Time*: Average CPU time in seconds.

Looking at Table 2 we see that

- Both *SC* and *GA* exhibit an excellent performance and consistently outperform all the other heuristics. In particular, the

Table 2
Performance of the heuristics.

	Class	B1				B2				B3			
	n	Gap	M_Gap	Best	Time	Gap	M_Gap	Best	Time	Gap	M_Gap	Best	Time
FFD	100	2.33	5.06	0	0.00	2.62	7.11	0	0.00	14.57	20.69	0	0.00
	200	1.95	5.11	0	0.00	1.92	4.79	0	0.00	16.02	21.97	0	0.00
	500	1.27	5.12	0	0.00	1.18	2.46	1	0.00	14.10	18.44	0	0.00
	1000	0.84	2.28	0	0.00	0.81	1.21	0	0.00	14.41	16.45	0	0.00
	2000	0.52	1.19	0	0.01	0.62	1.06	0	0.01	13.26	13.90	0	0.01
	AVG	1.38	5.12		0.00	1.43	7.11		0.00	14.47	21.97		0.00
BFD	100	2.31	5.06	0	0.00	2.73	7.11	0	0.00	14.41	20.69	0	0.00
	200	2.67	5.73	0	0.00	1.99	4.79	0	0.00	15.91	21.97	0	0.00
	500	1.21	5.12	0	0.00	1.38	2.46	0	0.00	14.00	18.44	0	0.00
	1000	1.09	2.56	0	0.00	0.79	1.21	0	0.01	14.38	16.45	0	0.00
	2000	0.83	1.55	0	0.01	0.62	1.06	0	0.01	13.36	14.19	0	0.01
	AVG	1.62	5.73		0.00	1.50	7.11		0.00	14.41	21.97		0.00
SSP 1	100	2.63	3.28	0	0.00	4.72	6.31	0	0.00	2.81	3.51	0	0.00
	200	2.43	3.04	0	0.01	4.99	6.31	0	0.01	3.12	3.83	0	0.02
	500	1.49	1.84	0	0.06	3.58	4.61	0	0.06	3.14	4.00	0	0.09
	1000	1.08	1.50	0	0.22	2.56	3.02	0	0.22	3.05	3.31	0	0.30
	2000	0.85	1.11	0	0.83	1.94	2.46	0	0.82	2.98	3.21	0	1.19
	AVG	1.70	3.28		0.22	3.56	6.31		0.22	3.02	4.00		0.32
SSP 2	100	2.65	3.28	0	0.00	4.72	6.31	0	0.00	11.01	14.29	0	0.00
	200	2.43	3.04	0	0.00	4.99	6.31	0	0.00	12.22	14.92	0	0.00
	500	1.49	1.84	0	0.02	3.58	4.61	0	0.02	12.63	15.49	0	0.02
	1000	1.08	1.50	0	0.08	2.56	3.02	0	0.07	12.54	13.65	0	0.08
	2000	0.85	1.11	0	0.28	1.94	2.46	0	0.27	12.77	13.50	0	0.27
	AVG	1.70	3.28		0.08	3.56	6.31		0.07	12.23	15.49		0.07
SSP 3	100	1.03	1.64	2	0.00	1.27	1.99	1	0.00	3.40	4.49	0	0.00
	200	1.12	1.96	2	0.00	1.42	2.25	2	0.00	3.93	5.07	0	0.01
	500	0.56	0.96	0	0.02	0.84	1.37	1	0.02	4.37	5.62	0	0.03
	1000	0.45	0.79	0	0.09	0.69	1.13	0	0.09	4.27	4.70	0	0.13
	2000	0.34	0.44	1	0.33	0.49	0.88	0	0.32	4.29	4.63	0	0.51
	AVG	0.70	1.96		0.09	0.94	2.25		0.09	4.05	5.62		0.14
SSP 4	100	2.63	3.28	0	0.00	4.72	6.31	0	0.00	2.86	3.51	0	0.00
	200	2.43	3.04	0	0.01	4.99	6.31	0	0.01	3.19	3.88	0	0.01
	500	1.49	1.84	0	0.07	3.58	4.61	0	0.07	3.09	3.96	0	0.06
	1000	1.08	1.50	0	0.25	2.56	3.02	0	0.25	3.01	3.27	0	0.25
	2000	0.85	1.11	0	0.93	1.94	2.46	0	0.93	2.94	3.14	0	0.93
	AVG	1.70	3.28		0.25	3.56	6.31		0.25	3.02	3.96		0.25
SC	100	0.85	1.41	8	0.23	1.03	1.83	10	3.77	2.22	3.08	9	0.09
	200	0.98	1.64	7	0.22	1.27	2.21	8	8.69	2.43	3.16	10	0.17
	500	0.46	0.80	8	3.85	0.71	1.21	9	12.47	2.56	3.33	10	0.54
	1000	0.37	0.72	9	5.95	0.62	1.03	9	19.78	2.51	2.83	10	1.83
	2000	0.32	0.41	9	12.89	0.44	0.82	10	21.41	2.50	2.69	10	6.73
	AVG	0.60	1.64		4.63	0.81	2.21		13.22	2.44	3.33		1.87
GEN	100	0.82	1.41	8	0.85	1.08	1.83	4	1.11	2.23	3.08	6	1.06
	200	0.95	1.64	9	1.96	1.32	2.21	4	1.99	2.57	3.49	0	1.98
	500	0.49	0.79	2	4.38	0.77	1.32	2	5.05	2.71	3.55	0	4.35
	1000	0.41	0.76	1	10.92	0.66	1.13	1	12.07	2.61	2.89	0	9.89
	2000	0.33	0.43	3	29.88	0.47	0.86	0	37.12	2.56	2.83	0	34.45
	AVG	0.60	1.64		9.60	0.86	2.21		11.47	2.54	3.55		10.35

average gap of *SC* over the 150 instances is 1.28% and the number of times it yields the best solution is 136. Interestingly, we see that the average gap over the 100 linear- and concave-cost instances is 0.70%. Moreover, we see that *GA* exhibits an average gap of 1.33% and delivers the best solution for 40 instances. However, and not surprisingly, these two effective heuristics require significantly longer CPU times than the four other simpler heuristics. Nevertheless, large-scale instances with up to 2000 items scarcely require more that 1 min CPU time.

- Despite its deceptive simplicity, *SSP* 3 was found to perform surprisingly well and achieves the best quality/efficiency trade-off. Indeed, it exhibits a very good average gap (1.78%) and is extremely fast since its average CPU time is 0.1 s only. Actually, *SSP* 3 consistently outperforms *SSP* 1, *SSP* 2 and *SSP* 4 on problem Classes B1 and B2 and performs reasonably well on Class B3.
- Overall, *SSP* 4 is lagged behind the best heuristics. Nonetheless, it is worth emphasizing that the major contribution of *SSP* 4 does lie in its ability to deliver high-quality solutions, but instead in the ability of its randomized variant to generate a set of input columns that render *SC* capable of performing extremely well.
- The performance of *SC* and *GA* is remarkably stable over all problem classes. Indeed, the maximal gaps over the 150 instances are 3.33% for *SC* and 3.55% for *GA*. On contrary, *SSP* 2 performs poorly on Class B3 with a maximal gap equal to 15.49%.
- Interestingly, we observe that the average gaps significantly depend on the problem classes. Indeed, we observe that the convex-cost instances of Class B3 often yield larger gaps for all heuristics. However, pushing our analysis a step further, we run *GA* on additional similarly generated small-sized instances (n ranging from 20 to 60 and $m = 3$). In so doing, we were able to compare the

Table 3
Performance of GA and SC heuristics with a fixed running time bound.

Time limit (s)	Class	B1						B2						B3					
	n	SC			GA			SC			GA			SC			GA		
		Gap	M_Gap	Best	Gap	M_Gap	Best	Gap	M_Gap	Best	Gap	M_Gap	Best	Gap	M_Gap	Best	Gap	M_Gap	Best
1	100	0.86	1.41	8	0.84	1.41	9	1.03	1.83	10	1.11	1.83	5	2.22	3.08	9	2.25	3.15	2
	200	0.98	1.64	7	0.96	1.64	9	1.27	2.21	9	1.32	2.21	3	2.43	3.16	9	2.56	3.34	1
	500	0.50	0.80	7	0.55	0.91	5	0.73	1.21	9	0.83	1.34	2	2.56	3.33	10	2.72	3.56	0
	1000	–	–	0	0.45	0.79	10	–	–	0	0.69	1.13	10	–	–	0	2.67	2.97	10
	2000	–	–	0	0.34	0.44	10	–	–	0	0.49	0.88	10	–	–	0	2.60	2.83	10
	AVG	–	–		0.63	1.64		–	–		0.89	2.21		–	–		2.56	3.56	
5	100	0.85	1.41	8	0.83	1.41	9	1.03	1.83	10	1.10	1.83	4	2.22	3.08	9	2.23	3.15	5
	200	0.98	1.64	7	0.95	1.64	9	1.27	2.21	8	1.31	2.21	4	2.43	3.16	10	2.59	3.32	0
	500	0.46	0.80	8	0.49	0.79	2	0.72	1.21	10	0.79	1.29	1	2.56	3.33	10	2.70	3.44	0
	1000	0.38	0.72	8	0.42	0.78	2	0.63	1.03	8	0.67	1.13	2	2.51	2.83	10	2.63	2.89	0
	2000	–	–	0	0.34	0.44	10	–	–	0	0.49	0.88	10	–	–	0	2.58	2.80	10
	AVG	–	–		0.61	1.64		–	–		0.87	2.21		–	–		2.55	3.44	
10	100	0.85	1.41	9	0.85	1.41	8	1.03	1.83	10	1.11	1.83	4	2.22	3.08	7	2.23	3.09	5
	200	0.98	1.64	8	0.97	1.64	8	1.27	2.21	7	1.30	2.21	4	2.43	3.16	9	2.59	3.54	1
	500	0.46	0.80	9	0.49	0.83	3	0.71	1.21	10	0.77	1.26	2	2.56	3.33	10	2.71	3.63	0
	1000	0.37	0.72	9	0.41	0.76	1	0.62	1.03	8	0.66	1.12	2	2.51	2.83	10	2.62	2.88	0
	2000	0.32	0.42	8	0.34	0.44	5	0.47	0.82	9	0.49	0.88	4	2.50	2.69	10	2.57	2.82	0
	AVG	0.60	1.64		0.61	1.64		0.82	2.21		0.87	2.21		2.44	3.33		2.54	3.63	
20	100	0.85	1.41	8	0.84	1.41	8	1.03	1.83	9	1.07	1.83	5	2.22	3.08	9	2.22	3.09	4
	200	0.98	1.64	8	0.96	1.64	9	1.27	2.21	8	1.32	2.21	4	2.43	3.16	9	2.56	3.45	1
	500	0.46	0.80	9	0.48	0.82	1	0.71	1.21	10	0.77	1.26	1	2.56	3.33	10	2.66	3.50	0
	1000	0.37	0.72	9	0.42	0.76	1	0.62	1.03	8	0.66	1.12	2	2.51	2.83	10	2.60	2.87	0
	2000	0.32	0.41	8	0.34	0.43	5	0.44	0.82	10	0.48	0.86	1	2.50	2.69	10	2.56	2.77	0
	AVG	0.60	1.64		0.61	1.64		0.81	2.21		0.86	2.21		2.44	3.33		2.52	3.50	

GA output with respect to the optimal solutions. For the sake of brevity, the detailed results are skipped. Interestingly, we found that although *GA* consistently delivers *optimal solutions* (there were only very few instances of Class B2 where optimality was not achieved), the deviation with respect to the lower bound was 0.95% for convex-cost instances. This might suggest that the (relative) large gaps that were observed for Class B3 are likely to result from the poor performance of the lower bound for this particular problem class.

5.3. Performance of GA and SC with a fixed running time

In order to get a more accurate picture of the actual performance of *GA* and *SC*, we investigated their respective efficacy with a running time bound. To that aim, we implemented for both of them the same time limit stopping criterion. We set this time limit to 1, 5, 10, and 20 s, respectively. The results are summarized in Table 3.

We see from Table 3 that if the time bound is set to a very short value (1 s) then *SC* fails to solve 1000- as well as 2000-item instances as well. Indeed, we observed that Phase 1 of *SC* (i.e. the column generation step) cannot be completed within this very short time bound. Similarly, SC fails to solve 2000-item instances if the time bound is set to 5 s. However, for both of these short time bounds *GA* delivers near-optimal solutions for all problem sizes. However, when the time bound is set to 10 s then *SC* solves all the instances and consistently outperforms *GA*. Indeed, we see that within this time bound, the number of times where *SC* yields the best solution (133) is much greater than the number of times *GA* does (47). Moreover, the same behavior is observed under a 20 s time bound. Hence, *GA* might be useful for deriving near-optimal solutions for large instances within very short CPU times. Otherwise, for medium size instances or if longer CPU times are allowed, then *SC* would be preferred since it consistently delivers high-quality solutions.

6. Conclusion

In this paper, we have addressed the deterministic one-dimensional bin-packing problem with unequal bin sizes and costs. We have proposed and analyzed the empirical performance of six heuristics for the *VSBPP*. These heuristics include: (i) four constructive methods that are based on the exact solution of subset-sum instances, (ii) a set covering heuristic where the columns are efficiently generated using a randomized bin-packing-based heuristic, and (iii) a GA that is based on a permutation coding of the solutions and that requires both solving shortest path problems and SSP. Our computational study, carried out on a large variety of problem instances, provides strong evidence of the efficacy of the set covering heuristic. Moreover, we realized that the GA performs very well and requires short CPU times. Finally, we found that a subset-sum-based heuristic is extremely fast and consistently outperforms two constructive heuristics from the literature.

Future work needs to be focused on the two-dimensional variable-sized bin-packing problem (2*D-VSBPP*). This problem, which is also referred to by the multiple stock sizes cutting stock problem, requires optimally packing a set of rectangular items into a set of rectangular bins with unequal sizes and costs [29]. Despite the practical industrial relevance of the 2*D-VSBPP* (e.g. in cutting of wood, glass, or metal plates), this problem has received scant attention in the operations research literature. We believe that an issue worthy of future investigation, is the development and analysis of highly effective heuristics for solving large-scale 2*D-VSBPP*.

References

[1] Friesen DK, Langston MA. Variable sized bin packing. SIAM Journal on Computing 1986;15:222–30.
[2] Murgolo FD. An efficient approximation scheme for variable-sized bin packing. SIAM Journal on Computing 1987;16:149–61.

[3] Coffman EG, Garey MR, Johnson DS. Bin packing with divisible item sizes. Journal of Complexity 1987;3:406–28.
[4] Kang J, Park J. Algorithms for the variable sized bin packing problem. European Journal of Operational Research 2003;147:365–72.
[5] Chu C, La R. Variable-sized bin packing: tight absolute worst-case performance ratios for four approximation algorithms. SIAM Journal on Computing 2001;30:2069–83.
[6] Monaci M. Algorithms for packing and scheduling problems, PhD thesis, University of Bologna, 2002.
[7] Belov G, Scheithauer G. A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths. European Journal of Operational Research 2002;141:274–94.
[8] Alves C, Valério de Carvalho JM. Accelerating column generation for variable sized bin packing problems. European Journal of Operational Research 2007;183:1333–52.
[9] Haouari M, Serairi M. Relaxations and exact solution of the variable sized bin packing problem. Computational Optimization and Applications, 2009, accepted.
[10] Caprara A, Pferschy U. Worst-case analysis of the subset sum algorithm for bin packing. Operations Research Letters 2004;32:159–66.
[11] Kellerer H, Pferschy U, Pisinger D. Knapsack problems. Berlin: Springer; 2004.
[12] Caprara A, Pferschy U. Modified subset sum heuristics for bin packing. Information Processing Letters 2005;96:18–23.
[13] Cullen F, Jarvis J, Ratliff D. Set partitioning based heuristics for interactive routing. Networks 1981;11:125–44.
[14] Kelly JP, Xu J. A set-partitioning-based heuristic for the vehicle routing problem. INFORMS Journal on Computing 1999;11:161–72.
[15] Marsten R, Shepardson F. Exact solution of crew scheduling problems using the set partitioning model: recent successful applications. Networks 1981;112: 167–77.
[16] Wedelin D. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. Annals of Operations Research 1995;57:283–301.
[17] Caprara A, Fischetti M, Toth P, Vigo D, Guida PL. Algorithms for railway crew management. Mathematical Programming 1997;79:125–41.
[18] Monaci M, Toth P. A set-covering-based heuristic approach for bin packing Problems. INFORMS Journal on Computing 2006;18:71–85.
[19] Beasley JE, Chu PC. A genetic algorithm for the set covering problem. European Journal of Operational Research 1996;94:392–404.
[20] Ceria S, Nobili P, Sassano A. A Lagrangian-based heuristic for large-scale set covering problems. Mathematical Programming 1998;81:215–28.
[21] Caprara A, Fischetti M, Toth P. A heuristic method for the set covering problem. Operations Research 1999;47:730–43.
[22] Haouari M, Chaouachi J. A probabilistic greedy search algorithm for combinatorial optimisation with application to the set covering problem. Journal of Operational Research Society 2002;53:792–9.
[23] Jakobs S. On genetic algorithms for the packing of polygons. European Journal of Operational Research 1996;88:165–81.
[24] Hopper E, Turton B. A genetic algorithm for a 2D industrial packing problem. Computers and Industrial Engineering 1999;37:375–8.
[25] Zhang DF, Chen SD, Liu YJ. An improved heuristic recursive strategy based on genetic algorithm for the strip rectangular packing problem. Acta Automatica Sinica 2007;33:911–6.
[26] Falkenauer E. A hybrid grouping genetic algorithm for bin packing. Journal of Heuristics 1996;2:5–30.
[27] Lima H, Yakawa T. A new design of genetic algorithm for bin packing, In: Evolutionary computation, 2003. CEC '03. The 2003 Congress on Evolutionary Computation, vol. 2, pp. 1044–9.
[28] Ruiz R, Maroto C, Alcaraz J. Two new robust genetic algorithms for the flowshop scheduling problem. Omega 2006;34:461–76.
[29] Pisinger D, Sigurd M. The two-dimensional bin packing problem with variable bin sizes and costs. Discrete Optimization 2005;2:154–67.