European Journal of Operational Research 224 (2013) 110–121

Contents lists available at SciVerse ScienceDirect

European Journal of Operational Research

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

Discrete Optimization

A goal-driven approach to the 2D bin packing and variable-sized bin packing problems

Lijun Wei [a], Wee-Chong Oon [c], Wenbin Zhu [b,*], Andrew Lim [a]

[a] Department of Management Sciences, City University of Hong Kong, Tat Chee Ave., Kowloon Tong, Hong Kong
[b] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
[c] School of InfoComm Technology, Ngee Ann Polytechnic, 535 Clementi Road, Singapore 599489, Singapore

ARTICLE INFO

Article history:
Received 11 August 2011
Accepted 7 August 2012
Available online 17 August 2012

Keywords:
Packing
2D bin packing
Goal-driven search
Best-fit heuristic

ABSTRACT

In this paper, we examine the two-dimensional variable-sized bin packing problem (2DVSBPP), where the task is to pack all given rectangles into bins of various sizes such that the total area of the used bins is minimized. We partition the search space of the 2DVSBPP into sets and impose an order on the sets, and then use a goal-driven approach to take advantage of the special structure of this partitioned solution space. Since the 2DVSBPP is a generalization of the two-dimensional bin packing problem (2DBPP), our approach can be adapted to the 2DBPP with minimal changes. Computational experiments on the standard benchmark data for both the 2DVSBPP and 2DBPP shows that our approach is more effective than existing approaches in literature.

1. Introduction

In the two-dimensional bin packing problem (2DBPP), a set of n *rectangles* with dimensions $w_i \times h_i$, $i = 1,\ldots,n$ are to be packed into rectangular *bins* with dimensions $W \times H$ so that the total area of the used bins is minimized (equivalent to minimizing the number of used bins). The rectangles can only be placed with sides parallel to the sides of the bins, and no two rectangles in the same bin may overlap. A useful generalization of the 2DBPP is the two-dimensional variable-sized bin packing (2DVSBPP), where we assume m types of bins with dimensions $W_j \times H_j$, $j = 1,\ldots,m$ are available. We can use up to N_j bins of type j. For both problems, we further assume that the rectangles cannot be rotated, and that all dimensions are integral. There are no other restrictions to the packing pattern (e.g., we do not impose the guillotine cut constraint). Both the 2DBPP and the 2DVSBPP are NP-hard since the one-dimensional bin packing problem, which is a special case of both problems, is NP-hard.

Both problems examined in this paper are two-dimensional packing problems (Lodi et al., 2002a). Under the improved typology of cutting and packing problems by Wäscher et al. (2007), the 2DBPP is called the Two-Dimensional Single Bin-Size Bin Packing Problem (2D-SBSBPP), while the 2DVSBPP is the Two-Dimensional Multiple Bin-Size Bin Packing Problem (2D-MBSBPP).

The 2DBPP has received significant attention from the research community; the survey by Lodi et al. (2002b) provides an overview of the problem and some of its variants. Several exact methods and lower bounds have been proposed for the problem, including by Dell'Amico et al. (2002), Boschetti and Mingozzi (2003a,b), Clautiaux et al. (2007), and Pisinger and Sigurd (2007). However, since the 2DBPP is NP-hard, heuristic solutions are likely to be required to solve the large problem instances encountered in practice. Examples of such approaches include tabu search (Lodi et al., 1999), guided local search (Faroe et al., 2003), a hybrid GRASP/VND approach (Parreño et al., 2010), and various other approaches based on greedy heuristics (Boschetti and Mingozzi, 2003b; Monaci and Toth, 2006; Zhang et al., 2011).

The 2DVSBPP is comparatively less well-studied, although a few related variants have been investigated, e.g., the 2DVSBPP with uniform bin costs and limits on bin types (Hopper and Turton, 2002a,b), or the variable bin cost 2DVSBPP (Pisinger and Sigurd, 2005). To the best of our knowledge, the only existing work that directly addresses the 2DVSBPP is by Ortmann et al. (2010), where the authors proposed two constructive heuristics called *stack ceiling* (SC) and *stack ceiling with re-sorting* (SCR).

This article proposes a goal-driven approach to solving the 2DBPP and 2DVSBPP. We partition the search space into subsets by objective value so that all solutions within a subset has the same cost. We then attempt to find a feasible solution in a subset. If we succeed, we proceed to search for a feasible solution in a subset with smaller cost to improve the known solution; if we fail, then we explore a subset with larger cost (where a feasible solution

* Corresponding author. Tel.: +852 64067667; fax: +852 34420188.
E-mail address: i@zhuwb.com (W. Zhu).

0377-2217/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ejor.2012.08.005

is expected to be easier to find). In essence, we are performing a binary search on the objective values of the solution subsets.

Our goal-driven approach (GDA) includes a tabu search using a greedy heuristic called SequencePack that attempts to pack as many rectangles as possible into a given set of bins. If SequencePack is not successful, we perform a local search called ForcePack on the best solution found so far that tries to pack all of the unpacked rectangles. We also designed a procedure called ImproveSol that attempts to reduce the total area of used bins in a feasible solution.

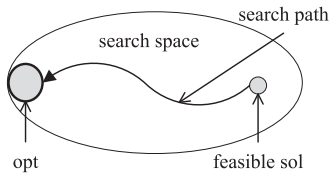
To evaluate the effectiveness of our approach on the 2DBPP, we tested it on the 500 benchmark test instances proposed by Berkey and Wang (1987) and Lodi et al. (1999). The results show that our approach was able to find solutions that required a fewer total number of bins on average over all instances than all existing approaches with 120 seconds of computation time. For the 2DVSBPP, the solutions obtained by our GDA approach were far superior to the existing heuristic approaches for all eight benchmark test sets proposed by Ortmann et al. (2010) with 120 seconds of computation time.

2. Goal-driven approach

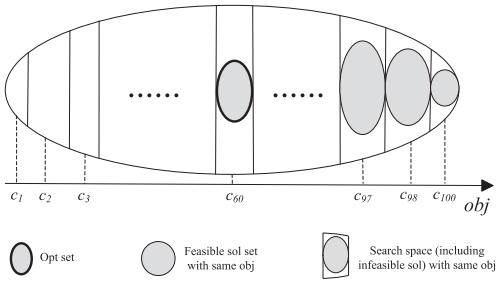
Given a starting feasible solution, the efficiency of an algorithm for an optimization problem depends on how quickly it approaches an optimal solution (see Fig. 1a). For many discrete optimization problems where the objective value is discrete, the number of objective values is small compared to the search space. For example, in the 2DBPP, the objective value is the number of used bins. In the 2DVSBPP, the number of different types of bins is often limited in reality, so the number of combinations of different bins is also small. Since there are far more solutions than different objective values, by the pigeonhole principle there will be a large number of different solutions with the same objective value in the search space. In fact, for any 2DBPP solution, we can get another solution with the same objective value by rearranging the order of the used bins or changing the packing configuration in any bin.

Conceptually, we can partition the search space of a problem into sets by the objective value of the solution. Fig. 1b gives an example of such a partition for the 2DVSBPP, where each set of the partition represents solutions where the used bins have the same area; note that more than one combination of bins may have the same total area. Since any member of a set is as good as others in the entire set, we only need to find one feasible solution for a set. Once we find such a solution, we can eliminate all sets whose objective value is equal or larger (assuming a minimization problem), which can reduce the search space dramatically. Our strategy is to generate search paths that identify one feasible solution per set.

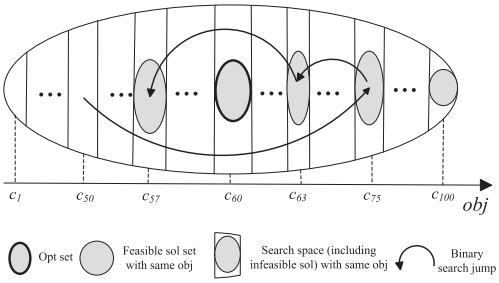
There are two challenges involved in implementing this idea. Firstly, we do not know the optimal objective value; this challenge can be overcome by using a binary search on objective values (Fig. 1c shows the sets considered in a binary search). Secondly, if the set we are considering contains no feasible solutions (i.e., the objective value of the set is smaller than optimal), then all search effort spent on this set is doomed to failure. We must therefore devise a method to avoid expending excessive computational effort on enumerating the elements in such sets. Using a lower bound is not an effective technique to resolve this issue since we may assume without loss of generality that the objective value of the set being considered is greater than the lower bound. In our approach, we limit the search effort spent on enumerating each set when trying to find a feasible solution using binary search. We then double the effort and invoke the binary search again in the next iteration. We continue doubling the effort until time limit is reached. Such a combination of binary search and iterative doubling has previously been applied to 2D strip packing by Wei et al. (2011).



(a) Optimization algorithm search path



(b) Search space partitioned by objective value



(c) Binary search on objective value

Fig. 1. Binary search on objective value.

The overall process of our goal-driven approach (GDA) for the 2DVSBPP is given in Algorithm 1. The values *LB* and *UB* define a range of objective values, i.e., we will attempt to find solutions with objective values in [*LB*,*UB*]. The value of *LB* is determined by finding a lower bound for the 2DVSBPP problem (line 1); we describe the details in Section 5. We eliminate the need for a true upper bound by guessing a value for *UB* (line 2), and if it turns out that the range [*LB*,*UB*] is too narrow to allow us to find a feasible solution, we simply enlarge the range (line 21). The outer while-loop (lines 4–22) iteratively doubles the search effort. The inner while-loop (lines 6–19) is the binary search that reduces the range of objective values by half in each iteration (lines 16–19).

Algorithm 1. Goal-driven approach for 2DVSBPP

```
Goal-Driven-Approach(R)
1  Compute lower bound LB
2  UB = LB × 1.2
3  iter = 1
4  while time limit not exceeded and LB ≠ UB
5      A = the set of all objective values in [LB, UB] attainable by
       some combination of bins, sorted in ascending order
6      while A ≠ ∅
7          obj = median value of A;
8          BCS = all combinations of bins whose total area is obj
9          for each combination BC ∈ BCS
10             Sort bins in BC in order of decreasing area
11             sol = TabuPack(BC, R, iter)
12             if all rectangles are packed in sol
13                 sol = ImproveSol(sol, iter);
14                 update UB and bestSol using sol;
15                 break
16         if solution was found in for-loop
17             A = all values in A less than UB
18         else
19             A = all values in A greater than obj
20     if no feasible solution found
21         UB = UB × 1.2
22     iter = iter * 2
23 return best solution found
```

For a given range [*LB*,*UB*], we first identify all objective values within the range that can be attained by some combination of bins, and sort these attainable objective values in ascending order (line 5). The attainable objective values can be computed by solving knapsack problems:

$$A = \left\{ z \middle| z = \sum_{i=1}^{m} W_j H_j x_j,\ \ LB \leqslant z \leqslant UB,\ \ x_j \in \{0,1,\ldots,N_j\},\ j = 1,2,\ldots,m \right\} \quad (1)$$

For a given target objective value obj, we find all combinations of bins whose total area is equal to obj (line 8) by solving the following knapsack problems:

$$BCS = \left\{ (x_1,x_2,\ldots,x_m) \middle| \sum_{i=1}^{m} W_j H_j x_j = obj,\ \ x_j \in \{0,1,\ldots,N_j\},\ j = 1,2,\ldots,m \right\} \quad (2)$$

Enumerating all elements of sets *A* and *BCS* requires exponential time in the worst case. However, in many practical applications the number of bin types is expected to be small. In such cases, the number of elements in *A* and *BCS* is small and we can enumerate them by simple brute force (it happens that the number of bin types is small in the benchmark test sets). When the number of bin types is large, we would have to adopt more advanced knapsack algorithms.

The innermost for-loop (lines 9–15) simply tries each combination of bins whose total area is obj and attempts to load all rectangles into the combination. For a given combination of bins, we first sort all bins by decreasing area (line 10), then invoke the TabuPack function to load as many rectangles as possible (in terms of total area); the parameter *iter* controls the total search effort spent by TabuPack. The details of TabuPack are provided in the following section. If all rectangles are packed, then we have found a feasible solution, whereupon we will try to further improve the solution by invoking the ImproveSol function (see Section 4).

For efficiency of implementation, we maintain a table that contains the sets *BCS* for each obj value that has been previously computed. Whenever we require the set *BCS* for a given obj value, we will first consult this table, and will only compute *BCS* if it has not been computed before.

Note that we can compute the set *A* incrementally. Let $g(a,b)$ denote the set *A* when $LB = a$, $UB = b$. We can verify from the definition that $g(a,b) = g(a,c) \cup g(c,b)$ for any c in $[a,b]$. That is, to compute *A* for a large interval, we can break the interval into smaller intervals, compute the respective sets *A* for the smaller intervals, and then merge the results. During the execution of Algorithm 1, we will need to compute *A* for a series of intervals $[LB,UB^{(1)}],[LB,UB^{(2)}],[LB,UB^{(3)}],\ldots$, where the superscripts denote the iteration. At the *k*th iteration, we use $maxUB^{(k)}$ to denote the largest interval encountered and $maxA^{(k)}$ to denote the corresponding set. These values can be updated as follows:

$$maxUB^{(k)} = \max(maxUB^{(k-1)}, UB^{(k)}) \quad (3)$$

$$maxA^{(k)} = maxA^{(k-1)} \cup g(maxUB^{(k-1)}, maxUB^{(k)}) \quad (4)$$

Once we have *maxA*, for any $UB \leqslant maxUB$ we can find the corresponding *A* by filtering.

Next, we would like to analyze the worst case scenario of our goal-driven-approach and discuss its robust implementation for practical use. It is possible that there is no feasible solution for a combination of bins with total area *obj*, yet there exists a feasible solution where the total area of the utilized bins is less than *obj*. It is also possible that no feasible solutions exist for every combination of bins with total area obj; we call such *obj* values *inadmissible*. In rare cases, there may exist a feasible solution with total area of bins *obj*′ less than an inadmissible obj; we call (*obj*′,*obj*) an *inverted pair*. Generally speaking, we expect the number of inverted pairs to be small, and when they do occur we expect the gap between *obj*′ and *obj* to be small (although it is possible to construct extreme counterexamples).

The worst case scenario occurs when there exists a feasible solution with total area *z*, but in the *k*th iteration our binary search tries a sequence of inadmissible targets $obj^{(k,1)} < obj^{(k,2)} < obj^{(k,3)} \ldots$ that are all greater than *z* and returns a feasible solution whose total area of bins equals *UB*. Subsequent iterations of binary search will go through the same sequence of inadmissible targets and return the same feasible solution since *UB* has not changed. Hence, our algorithm stalls. The probability of stalling is expected to be

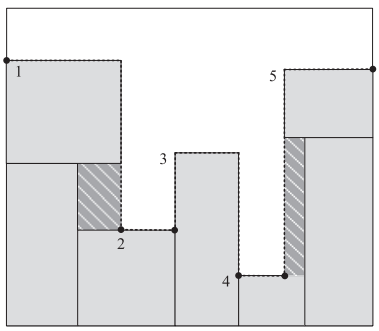


Fig. 2. Example of skyline.

small, because if the kth iteration either stops without finding any feasible solution or with a feasible solution whose total area is less than UB, the next iteration will try a different sequence of target areas. The chance of all new target areas being inadmissible is very slim if not zero.

When stalling does occur, it can be easily detected and we can simply introduce a small random perturbation in the next iteration once we detect it. The purpose of the random perturbation is to ensure the sequence of targets examined in the next iteration is different from the current iteration, so that the chance of all targets being inadmissible is greatly reduced.

3. TabuPack packing procedure

Our TabuPack packing procedure given in Algorithm 2 is a straightforward tabu search, which tries to pack the rectangles into the given bin combination as much as possible. It uses a procedure called SequencePack as a subroutine, which attempts to pack a given sequence of rectangles into a sequence of bins using a best-fit rule. There are two possible implementations of best-fit: (1) items are loaded strictly according to the order in the sequence and each item is placed at the best position and (2) all possible ways of placing an item at a position are evaluated, and the best placement (i.e., an item and a corresponding position) is selected and executed at each step. Our SequencePack implements both strategies: the first ff items will be placed using the first best-fit rule and the remaining items will be placed using the second best-fit rule. When ff is set to $|R|$, we will load all items using the first best-fit strategy; when ff is set to 0, we will load all items using the second best-fit strategy; and when ff is set to $|R|/2$, we will use both strategies. We applied tabu search three times, once for each value of $ff \in \{0, \lfloor |R|/2 \rfloor, |R|\}$. The initial sequence for all three tabu searches is obtained by sorting the set of rectangles R by decreasing area.

For each ff value, we perform at most $iter$ iterations of tabu search. Given the current sequence seq at the beginning of an iteration (lines 6–15), we randomly select two rectangles b_i and b_j. If (b_i, b_j) is not in the tabu list, we swap them to produce a new sequence. In this manner, we generate up to 10 non-tabu sequences. From these 10 non-tabu sequences, we select the one that produces the solution with the highest area utilization, and insert the swap into the tabu list; for the next $3n$ iterations, where n is the number of rectangles in R, this swap is forbidden.

If the tabu search fails to produce a solution where all rectangles are packed, then we use a local search subroutine called ForcePack that makes a final attempt to pack all the unpacked rectangles in the best solution found for that sequence and ff value; preliminary experiments showed that ForcePack was only effective when the solution has 6 or more bins. At any stage, if a solution is found that packs all rectangles into the bins, then we immediately terminate the procedure and return the result. Otherwise, we return the best solution found in terms of area utilization.

Algorithm 2. TabuPack procedure

```
TabuPack(B, R, iter)
 1  for each ff ∈ {0, ⌊|R|/2⌋, |R|}
 2      Seq = Sort rectangles in R in order of decreasing area
 3      sol = SequencePack(Seq, B, ff)
 4      if all rectangles are packed in sol then return sol
 5      bestSol = sol
 6      for i = 1 to iter
 7          Generate 10 non-tabu sequences {Seq_1, ..., Seq_10} from Seq by swapping two rectangles
 8          for each s ∈ {Seq_1, ..., Seq_10}
 9              sol = SequencePack(s, B, ff)
10              if all rectangles are packed in sol then return sol
11              Update bestSol if area utilization of sol is higher than bestSol
12          Let Seq_x ∈ {Seq_1, ..., Seq_10} be the sequence that produced the highest area utilization packing
13          Let (b_i, b_j) be the rectangles swapped to produce Seq_x from Seq
14          Add (b_i, b_j) to the tabu list for the next 3n iterations where n is the number of rectangles in R
15          Seq ← Seq_x
16      if number of bins in bestSol > 5
17          sol = ForcePack(bestSol, iter)
18          if all rectangles are packed in sol then return sol
19  return best solution found
```

3.1. SequencePack subroutine

Our SequencePack procedure takes as input a sequence of rectangles Seq, a set of bins B that we maintain in a fixed sequence, and an integer parameter ff. The purpose of our SequencePack subroutine is to pack a sequence of rectangles into the given sequence of bins. It is adapted from the 2DRPSolver heuristic by Wei et al. (2011) for the 2D rectangle packing problem. The 2DRPSolver heuristic is a skyline-based best-fit algorithm to pack a set of rectangles into a single bin, which is in turn an extension of the skyline-based best-fit algorithm by Burke et al. (2004). The packing pattern in each bin is represented by a skyline, and the candidate positions to place a rectangle are given by the "concave" points on the skyline (see Section 3.1.1 for more details). The procedure then repeatedly selects a placement (i.e., a rectangle along with a position in some bin) according to a fitness measure and packs the selected rectangle at that position until no feasible placements remain (either because

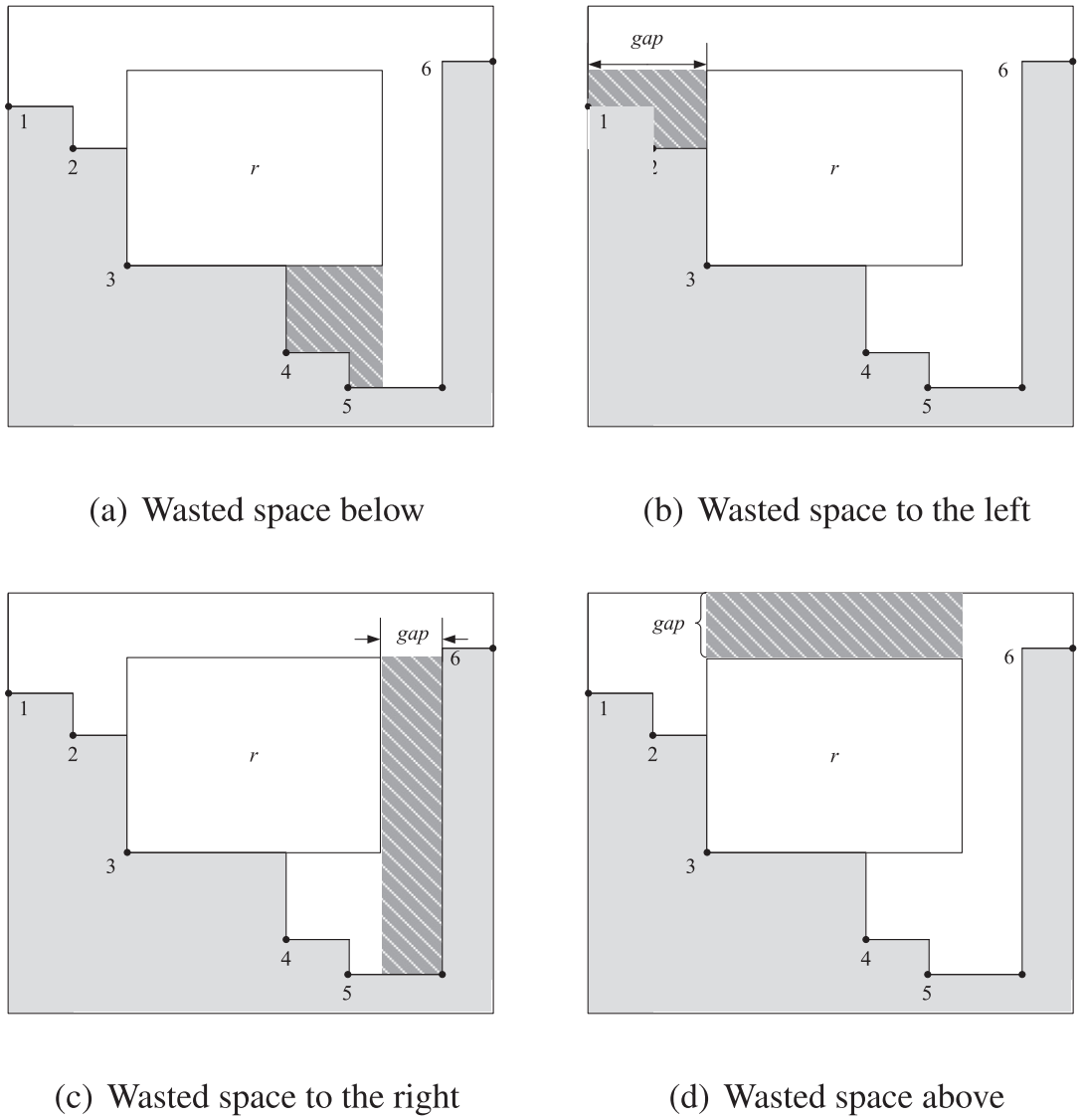


Fig. 3. Wasted local space.

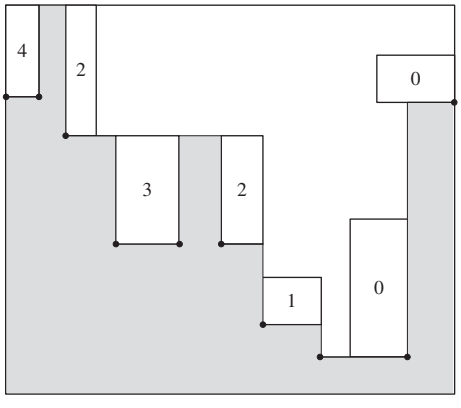


Fig. 4. Fitness numbers.

there are no more rectangles, or no more positions can accommodate any rectangles).

We propose two fitness measures for this purpose. The first fitness measure (Eq. (5)) tries to load rectangles according to the order given in the sequence at the best position whenever possible. The second fitness measure (Eq. (6)) de-emphasizes the sequence and focuses more on loading the best rectangle to the best position.

We define a function f to measure the fitness of packing a rectangle r at a candidate position p. Our fitness function $f(r,p)$ produces a vector that consists of seven components. We say that (r,p) is fitter than (r',p') if $f(r,p)$ is lexicographically smaller than $f(r',p')$ (i.e., we compare the two vectors component by component, and the order of the first different component decides the order of the two vectors). The seven components of $f(r,p)$ are:

$$f(r,p) = (ord(r), onlyFit(r,p), waste(r,p), -fn(r,p), binOrd(p), y(p), x(p)) \quad (5)$$

where

1. $ord(r)$: the order of rectangle r in the input sequence.
2. $onlyFit(r,p)$: if rectangle r is the only rectangle remaining that can fit onto p, this value is 0; otherwise, this value is 1.
3. $waste(r,p)$: the total amount of adjacent wasted space (where no other rectangle can be placed) that is created if r is placed at p. A more detailed explanation is given in Section 3.1.2.

4. $fn(r,p)$: the number of sides of the rectangle that exactly matches the segment it is touching in the skyline, called its *fitness number* (see Section 3.1.3 for details).
5. $binOrd(p)$: the order of the bin containing position p in the ordered set B.
6. $y(p)$: the y-coordinate of position p.
7. $x(p)$: the x-coordinate of position p.

During the packing process, we use the above fitness function to select the first *ff* rectangles and their corresponding candidate positions. The remaining $n - ff$ rectangles (and their corresponding candidate positions), however, are selected by a slightly different fitness function:

$$f'(r,p) = (onlyFit(r,p), waste(r,p), -fn(r,p), ord(r), binOrd(p), y(p), x(p)) \quad (6)$$

3.1.1. Skyline representation of a packing pattern

We represent a current packing pattern by a rectilinear *skyline*, which is the contour of a set of consecutive vertical bars. It can be expressed as a sequence of k horizontal line segments $(s_1, s_2, \ldots, s_k)$ satisfying the following properties: (1) the y-coordinate of s_j is different from the y-coordinate of s_{j+1}, $j = 1,\ldots,k-1$ and (2) the x-coordinate of the right endpoint of s_j is the same as the x-coordinate of the left endpoint of s_{j+1}, $j = 1,\ldots,k-1$. Fig. 2 gives an example of a skyline, where each line segment s_j is labeled as j at its left endpoint. The initial empty packing pattern is represented by a single line segment corresponding to the bottom of the bin.

3.1.2. Local waste

Our third component of fitness function is the amount of *adjacent waste space*. This is calculated as the total volume of wasted space of the four types given in Fig. 3a–d. Let *w_min* and *h_min* be the minimum width and height, respectively, of the remaining rectangles excluding r. The shaded area in case (a) is always considered wasted space. The shaded areas in cases (b) and (c) are considered wasted if the length of the *gap* is less than *w_min*. Finally, the shaded area in case (d) is wasted if the *gap* is less than *h_min*. This rule is motivated by the natural assumption that if the amount of wasted space is minimized at every stage of the process, then the remaining unplaced rectangles will be more likely to be placeable.

3.1.3. Fitness numbers

Our fourth component of the fitness function is the *fitness number*. The fitness number of a placement is the number of sides of the rectangle that exactly matches the segment it is touching in the skyline. The bottom side of a rectangle is an exact match if its width is equal to the length of s_j. The left (resp. right) side of a rectangle is an exact match if its height is equal to the y-coordinate of s_{j-1} (resp. s_{j+1}) minus the y-coordinate of s_j. Any side of a rectangle that touches the left, right or bottom side of the sheet is not considered an exact match unless it fills all the remaining space on that side. However, when the top edge of a placed rectangle touches the top of the sheet, this is considered an exact match. The fitness number can be either 0, 1, 2, 3 or 4, as shown in Fig. 4, where the number in each rectangle depicted is its fitness number. This rule favors placements that result in a "smoother" skyline that contains fewer line segments, which is more likely to allow the placement of larger rectangles without producing wasted space.

3.2. Squeeze operator

Our ForcePack procedure uses a neighborhood operator called Squeeze as shown in Algorithm 3. Given a solution *sol* containing some unpacked rectangles, this operator attempts to pack the unpacked rectangles as much as possible into some subset B' of the used bins.

Algorithm 3. Squeeze operator

```
Squeeze (sol, iter, B')
1  Unload bins B' in sol and add all the rectangles in B' to U
2  partSol = TabuPack (B', U, min{iter, |U|})
3  return combined solution of sol and partSol
```

The solution *sol* can be represented by two sets: the set $B = (B_1, B_2, \ldots, B_k)$ is the sequence of used bins, and the set $R = \{U, R_1, \ldots, R_k\}$ is the set of rectangles, where R_i, $i = 1,\ldots,k$ is the set of rectangles packed into bin B_i, and U is the set of unpacked rectangles. Given a subset of used bins $B' \subset B$, the Squeeze function first unloads all rectangles from the bins in B' from *sol* and appends them to the set of unpacked rectangles set U, then invokes the TabuPack algorithm to pack the unpacked rectangles as much as possible into the bins in B'; the number of iterations of tabu search is set to be either the current *iter* value or the number of rectangles in the sub-problem, whichever is lower. This sub-solution is then re-combined with *sol* and returned as the output.

3.3. ForcePack subroutine

After performing the tabu search for a particular *ff* value, if a solution that packs all rectangles is not found, then we perform a procedure called ForcePack on the best solution *bestSol* found for that *ff* value if *bestSol* consists of more than five bins. This procedure can be considered a local search whose objective is to pack all of the unpacked rectangles into the existing bins in the solution. It is similar to the fourth operator used in the improvement phase of the GRASP/VND approach for the 2DBPP by Parreño et al. (2010).

The ForcePack procedure is given by Algorithm 4. In iteration i, we consider all sub-problems consisting of either one bin B_i, or two bins (B_i and one of $B_1,\ldots,B_{i-1}$). For each sub-problem, we invoke the Squeeze procedure. We retain the resulting solution if it is superior and discard it otherwise. This process is repeated until either all rectangles have been packed or all combinations have been considered without an improvement being discovered.

Algorithm 4. ForcePack procedure

```
ForcePack (sol, iter)
1  repeat
2    improved = false
3    for i = 1 to k
4      for j = 1 to i
5        newsol = Squeeze (sol, iter, B_i ∪ B_j)
6        if all rectangles are packed in newsol then return
   newsol
7        if area utilization of newsol> area utilization of sol
8          sol = newsol
9          improved = true
10 until improved == false
11 return sol
```

Fig. 5 gives an example of how ForcePack works. In the initial solution shown in Fig. 5a, rectangle r is not packed. At this point, we may invoke the Squeeze operator with $B' = (B_1, B_2)$ (Fig. 5b). If this is successful, e.g., as given in Fig. 5c, then we have a solution that packs all rectangles.

4. ImproveSol post-improvement procedure

When a feasible solution *sol* has been found (i.e., all rectangles are packed in the given bins in *sol*), we use a procedure called ImproveSol (Algorithm 5) that attempts to reduce the area of the used bins in the solution. The ImproveSol procedure considers each bin B_i in reverse order and calls the procedure DeleteOrReplaceOneBin, which

attempts to either pack all rectangles in B_i into other bins used in the current solution or replace B_i by a smaller bin. In the procedure DeleteOrReplaceOneBin, we first unload the rectangles in B_i, and then try to pack all the unpacked rectangles as much as possible into other bins used in the current solution. If all the rectangles are packed, then we have sucessfully deleted B_i; otherwise, we try to find the smallest bin that can contained the remaining unpacked rectangles.

Algorithm 5. ImproveSol and DeleteOrReplaceOneBin procedure

```
ImproveSol(sol, iter)
1  for i = k to 1
2      sol = DeleteOrReplaceOneBin(sol, iter, i)
3  return sol

DeleteOrReplaceOneBin(sol, iter, i)
1  oldSol = sol
2  sol = unload B_i in sol
3  for j = 1 to k
4      if i ≠ j
5          newSol = Squeeze(sol, iter, B_j)
6          if all rectangles are packed in newSol then return newSol
7          if area utilization of newSol > area utilization of sol
8              sol = newSol
9  C = the set of bin types whose area is less than B_i in increasing order of area
10 for each each bin type b ∈ C
11     partSol = TabuPack(b, U, min{iter, |U|})
12     if all rectangles are packed in partSol then return combined solution of sol and partSol
13 return oldSol
```

The DeleteOrReplaceOneBin procedure is given in Algorithm 5. We first unload the bin B_i, and then attempt to pack the unpacked rectangles into each of the other bins B_j, $j = 1,\ldots,k$, $j \neq i$. This is done by invoking the Squeeze procedure on the sub-problem consisting of B_j. If the total area of the unpacked rectangles is reduced as a result, we continue the search with this solution, otherwise we discard it. If all rectangles in B_i are eliminated at any stage, we return this solution; however, if we are unable to eliminate all rectangles in B_i after considering all bins $B_j \neq B_i$, then we will try to pack all the unpacked rectangles into as small a bin as possible. If we successfully place the unpacked rectangles into a bin b whose area is less than B_i, we replace B_i by b and return this solution; otherwise, the original solution will be returned.

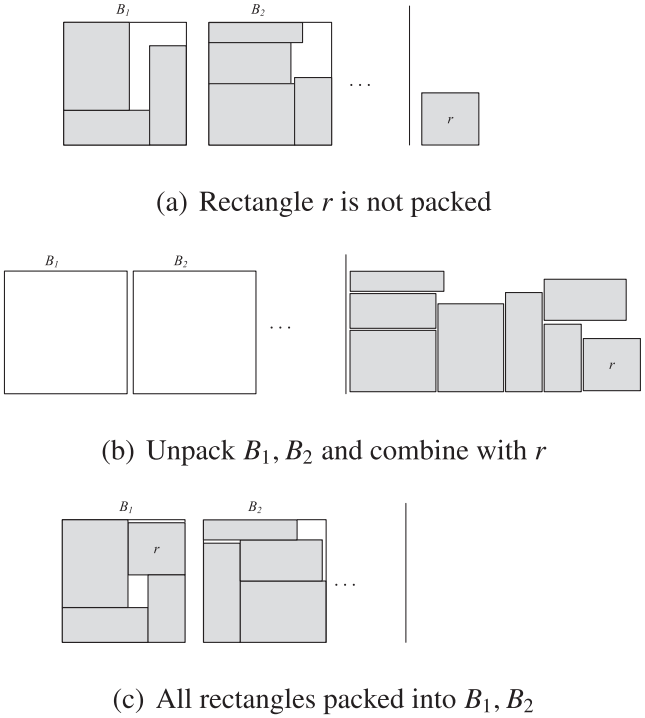
Fig. 6 shows an example of the DeleteOrReplaceOneBin procedure. Given the current bin B_i, we first unpack both B_i and B_1 (Fig. 6b), and then attempt to pack these rectangles into B_1. If the area of the remaining rectangles is reduced (like in Fig. 6c), then we continue with this solution, otherwise we discard it. The procedure continues with the unpacking of B_2 (Fig. 6d), and so on. If all rectangles in B_i are eventually packed into other bins used in the current solution (Fig. 6e), then we have successfully eliminated B_i.

5. Computing the lower bound

For the 2DBPP, there are several lower bound measures that have been proposed in existing literature. The best lower bounds for the 2DBPP benchmark instances are summarized at DEIS (2012), which are the values we used for our experiments.

For the 2DVSBPP, recall that there are m types of bins $B_1, B_2, \ldots, B_m$ and we are to pack all rectangles in the set R. We first compute values L_j and U_j for each type of bin B_j, such that in any solution at most U_j and at least L_j bins of type j would be required.

The value of L_j is computed as follows. Let $\overline{R}_j \subseteq R$ be the set of rectangles that can *only* be packed into bins of type j, i.e., these rectangles cannot fit into any other bin type. This allows us to de-

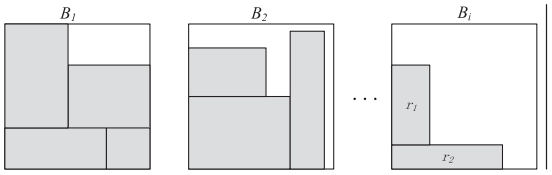


(a) Rectangle r is not packed

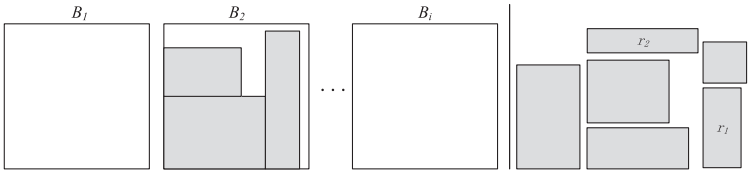
(b) Unpack B_1, B_2 and combine with r

(c) All rectangles packed into B_1, B_2

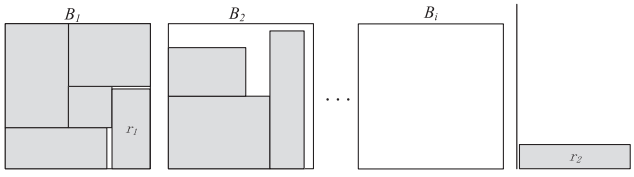
Fig. 5. Example of ForcePack heuristic.



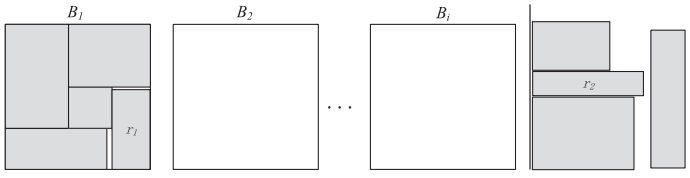
(a) Rectangles r_1 and r_2 are in B_i



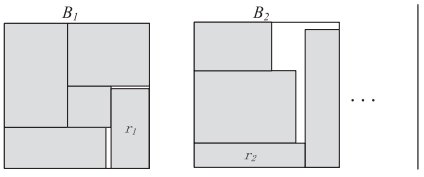
(b) Unpack R_1 and combine with r_1, r_2



(c) Repack rectangles into B_1



(d) Unpack R_2 and combine with r_2



(e) Repack rectangles into B_2

Fig. 6. Example of DeleteOrReplaceOneBin procedure.

rive $L'_j = \lceil (\sum_{i \in \overline{R}_j} w_i \times h_i)/(W_j \times H_j) \rceil$, which is the minimum number of bins of type j required such that there is sufficient area to contain all rectangles in $\overline{R}_j$. Additionally, we also find the set $\overline{R}'_j \subseteq \overline{R}_j$ containing all rectangles in $\overline{R}_j$ that have a magnitude of more than half the bin on both dimensions, i.e., $\overline{R}'_j = \{r_i \in \overline{R}_j : w_i > \frac{W_j}{2}$ and $h_i > \frac{H_j}{2}\}$. Obviously, no two rectangles in $\overline{R}'_j$ can be in the same bin, so $L_j \geqslant |\overline{R}'_j|$. We set $L_j = \max\left\{L'_j, \left|\overline{R}'_j\right|\right\}$.

To compute U_j, we find the set $R_j \subseteq R$ containing all rectangles that can be packed into bins of type j, and then solve a two-dimensional bin packing problem using only bins of type j and the set of rectangles R_j. To do so, we first set $U_j = \lceil (\sum_{i \in R_j} w_i \times h_i)/(W_j \times H_j) \rceil$, and then invoke TabuPack on R_j using U_j bins of type B_j and *iter* = 1. If this is unsuccessful, we add an additional bin of type B_j and repeat the process until all rectangles are successfully packed or until the maximum number of bins N_j (as given in the input) is reached.

Having found the values of L_j and U_j, our overall *LB* is derived by solving the following knapsack problem, where x_j is the number of bins of type j used:

Table 1
Computational results for **2DBPP**.

Class	TS3	GLS	HBP	SCH	GRASP	BS-EPSD	GDA			
							$\sum \#bin_{avg}$	#opt	TTB_{avg}(s)	TTB_{max}(s)
1	101.5	100.2	99.9	**99.7**	**99.7**	**99.7**	**99.7**	46	0.13	1.67
2	13.0	**12.4**	**12.4**	**12.4**	**12.4**	**12.4**	**12.4**	50	0.00	0.01
3	72.6	70.2	70.3	69.6	69.6	**69.5**	69.6	41	3.63	109.7
4	12.8	12.5	12.5	12.4	12.3	12.3	**12.0**	49	1.80	86.5
5	91.3	90.2	89.9	89.3	89.3	**89.2**	89.3	40	2.22	35.8
6	11.5	11.4	11.3	11.2	11.2	**11.1**	**11.1**	47	1.44	70.0
7	84.0	83.4	83.2	**82.7**	82.8	**82.7**	**82.7**	36	1.40	11.6
8	84.4	84.1	83.9	83.6	**83.4**	**83.4**	**83.4**	42	0.78	7.60
9	213.1	**213.0**	**213.0**	**213.0**	**213.0**	**213.0**	**213.0**	50	0.22	0.99
10	51.8	51.0	51.1	50.4	50.4	50.3	**50.2**	38	2.35	89.0
Total	736	728.4	727.5	724.3	724.1	723.6	**723.4**	439		

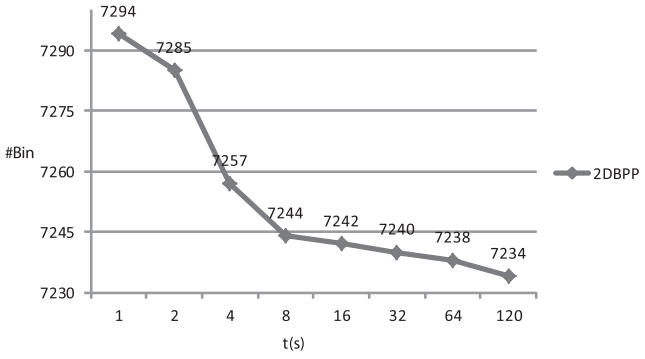


Fig. 7. Convergence behavior on 2DBPP.

$$LB: \quad \min \sum_{j=1}^{m} W_j \cdot H_j \cdot x_j \tag{7}$$

$$L_j \leqslant x_j \leqslant U_j \quad 1 \leqslant j \leqslant m \tag{8}$$

$$\sum_{j=1}^{m} W_j \cdot H_j \cdot x_j \geqslant \sum_{i=1}^{n} w_i \cdot h_i \tag{9}$$

$$x_j : integer \tag{10}$$

Note that when calculating U_j, if $R_j = R$, then we have found a feasible solution to the 2DVSBPP. In this case, we first invoke the ImproveSol procedure on this solution, and then set the overall upper bound *UB* to be the total area of the bins in this solution if it is better. In addition, we omit the step where the upper bound is set to be 1.2 times the lower bound (line 2 in Algorithm 1). We also make use of the values of L_j and U_j when calculating the combinations of bins for the binary search, i.e., $x_j \in \{L_j, L_j + 1, \ldots, U_j\}$ in Eqs. (1) and (2).

6. Computational experiments

We tested our GDA approach on existing benchmark test data for both the 2DBPP and 2DVSBPP. Our algorithms were written as sequential algorithms in C++ and compiled using GCC 4.1.2 with no explicit multi-threading utilized. Our experiments were performed on an Intel Xeon E5430 with a 2.66 gigahertz (Quad Core) CPU and 8 gigabytes RAM running the CentOS 5 linux operating system.

For all experiments, we set a computational time limit of 120 CPU seconds or until the first feasible solution has been found, whichever is longer. We define the first feasible solution to be the first solution that packs all rectangles in the original problem (found either by the TabuPack procedure, or when computing the lower bound for 2DVSBPP) followed by the ImproveSol post-improvement heuristic. The detailed results of all experiments can be found online at Wei and Zhu (2011).

6.1. Experiments for 2DBPP

For the 2DBPP, we evaluated our approach on 500 standard benchmark instances divided into 10 classes. Classes 1–6 were generated by Berkey and Wang (1987), and Classes 7–10 were generated by Lodi et al. (1999). Each class of instances is further divided into five groups, and every group consists of 10 instances with same number of rectangles; the number of rectangles per instance in the five groups are 20, 40, 60, 80, and 100, respectively. All instances can be found at DEIS (2012).

The computational results are summarized in Table 1; more detailed results are given in Appendix B in the online supplements. We compared GDA with the following existing approaches (the computational environments for these approaches are summarized in Appendix A in the online supplements):

- *GRASP*: the hybrid GRASP/VND algorithm by Parreño et al. (2010).
- *SCH*: the Set Cover based Heuristic algorithm by Monaci and Toth (2006).
- *GLS*: the Guided Local Search algorithm by Faroe et al. (2003).
- *TS3*: the Tabu Search algorithm by Lodi et al. (1999) for 2D bin packing.
- *HBP*: the HBP heuristic by Boschetti and Mingozzi (2003b).
- *BS-EPSD*: the heuristic based on space defragmentation by Zhang et al. (2011).

The values in each entry for these approaches is sum of the average number of bins used for each group within the class; this value is listed under the $\sum \#bins_{avg}$ column for GDA. Additionally, for GDA the column *#opt* gives the number of optimal solutions found for each class (out of 50) where the number of bins in the solution is equal to the lower bound. For each instance, we record the time-to-best, that is, the time when the best solution is first found. The average time-to-best over a class of instances is reported in the column $TTB_{avg}(s)$. The maximum time-to-best over a class of instances is reported in the column $TTB_{max}(s)$.

The results show that GDA found the best or joint-best solutions on average compared to all existing approaches for 8 out of the 10 classes, and its average utilization of 723.4 over all classes is better than all other approaches. Although the amount of improvement by GDA over existing approaches is small (only two bins in total over all classes), the 2DBPP is a very well-studied problem, and improvements have become increasingly difficult to attain.

Table 2
Summary of benchmark data for 2DVSBPP.

Class	#Instances	Bin				Rectangle			
		m	W	H	$W \times H$	n	w	h	$w \times h$
P1	1	3	272–528	768–768	208,896–405,504	2900	95–272	137–329	26,695–79,631
P2	1	2	48–60	96–108	2880–6480	900	12–28	18–30	216–840
M1	5	6	10–20	10–40	100–600	100,150	1–9	1–9	1–81
M2	5	6	30–60	30–120	900–5400	100,150	3–28	3–28	9–784
M3	5	6	30–60	30–120	1200–6000	100,150	3–30	3–30	9–900
PS	500	6	5–300	5–300	25–90,000	20,40,60,80,100	1–100	1–100	1–10,000
Nice	170	2–6	153–1000	165–1000	46,025–746,000	25,50,100,200,500	13–465	14–473	552–172,220
Path	170	2–6	188–1000	205–1000	51,728–769,000	25,50,100,200,500	1–500	1–500	1–211,640

Table 3
Computational results for **2DVSBPP**.

Class	SC	SCR	GDA (first)		GDA (best)							
			pu_{avg}	TTF(s)	pu_{avg}	$\#bin_{avg}$	$\#bin_{max}$	rg_{avg}	rg_{max}	$\#opt$	TTB_{avg}(s)	TTB_{max}(s)
P1	84.4	84.4	**89.7**	225.9	**89.7**	390.0	390.0	11.5	11.5	0	225.9	225.9
P2	90.0	90.0	**93.3**	8.97	**93.3**	64.0	64.0	6.7	6.7	0	8.97	8.97
M1	95.5	95.5	**96.9**	0.07	**98.4**	6.2	7.0	0.0	0.0	5	0.10	0.15
M2	90.8	90.8	**92.9**	0.12	**93.6**	7.8	8.0	6.4	9.6	1	0.57	2.35
M3	93.9	93.9	**96.0**	0.20	**96.0**	9.8	10.0	3.6	7.5	1	0.20	0.27
PS	82.1	81.9	**84.9**	0.18	**89.8**	16.0	78.0	10.4	41.9	78	25.4	119.8
Nice	83.1	82.9	**84.9**	0.72	**95.5**	4.8	10.0	5.2	50.0	25	28.4	119.8
Path	85.3	85.2	**87.1**	0.82	**95.7**	5.2	15.0	4.8	30.0	23	29.4	119.8

Fig. 7 plots the total number of bins used for all instances by GDA over time; note that the time values on the x-axis increase exponentially up to 120 seconds. We see that the majority of the improvement to the solutions occur within the first 8 seconds of computation. Beyond this point, the solutions found by GDA continue to improve but at a slower rate. In particular, in terms of the total number of bins, GDA outperforms the published results of SCH (Monaci and Toth, 2006) after 16 seconds, GRASP (Parreño et al., 2010) after 32 seconds, and BS-EPSD (Zhang et al., 2011) after 120 seconds on our machine.

6.2. Experiments for 2DVSBPP

We separate the 2DVSBPP benchmark data into eight classes:

- *P1, P2*: Wang (1983) proposed two instances P1 and P2, which we consider as two separate data classes having one instance each. These instances were based on industry data, and are characterized by the large number of rectangles to be packed (2900 for P1 and 900 for P2). Note that the granularity of the rectangles in these instances are up to one-eighth of an inch; since our approach requires integer dimensions, we converted these instances to equivalent integral instances by multiplying all values by 8 prior to computation.
- *M1, M2, M3*: Hopper (2000) and Hopper and Turton (2002c) generated 15 instances divided equally into three classes. Classes M1 and M2 have 100 rectangles each, and Class M3 has 150 rectangles.
- *PS*: Pisinger and Sigurd (2005) generated 500 new 2DVSBPP instances with variable bin costs from the existing 500 2DBPP benchmark instances. For each 2DBPP instance involving a bin of dimensions $W \times H$, the corresponding new 2DVSBPP instance includes five additional bin types, where the dimensions of each additional bin are uniformly independently selected from the range $[W/2, W] \times [H/2, H]$. These instances were subsequently adapted for the 2DVSBPP by Ortmann et al. (2010) by removing the variable bin costs. Note that no limits on the number of bins N_j for each bin type j were specified for this data set.
- *Nice, Path*: These two data classes were generated by Ortmann et al. (2010) based on the method for generating data sets for rectangular placement problems by Valenzuela and Wang (2001). Each class consists of 170 instances with 2–6 bin types and 25–500 rectangles. Since these instances were generated by cutting bins into rectangles, they have optimal solutions with 100% area utilization.

The characteristics of the benchmark test data are summarized in Table 2. The test instances are provided by Wei and Zhu (2011) and are available online. We compare our approach with the stack ceiling (SC) and stack ceiling with re-sorting (SCR) algorithms by Ortmann et al. (2010), which are the only existing 2DVSBPP approaches in literature to the best of our knowledge; both algorithms are simple constructive heuristics. The results are summarized in Table 3; more detailed results can be found in Appendix C in the online supplements. Columns *SC* and *SCR* give the average packing area utilization for each class for the SC and SCR algorithms, respectively; the corresponding values achieved by GDA are reported under the columns pu_{avg}.

Under the heading *GDA (first)*, we give the average area utilization (pu_{avg}) and the time required by GDA to compute the first feasible solution in CPU seconds ($TTF(s)$). The columns under the heading *GDA (best)* provide the information for the best solution found. For classes *Nice* and *Path*, which have known perfect optimal solutions, the column *#opt* is the number of solutions with 100% area utilization; for the other classes, *#opt* is the number of solutions with area utilization equal to the computed lower bound as described in Section 5. The column $\#bin_{avg}$ and $\#bin_{max}$ report the average and maximum number of bins in the solutions of instances in a class. We also computed the relative gap between our final solutions and the lower bounds; the average over a class is

reported in column rg_{avg} and the maximum over a class is reported in column rg_{max}.

The results show that the average quality of the first solution found by GDA is superior to those of SC and SCR for all classes (although there are some individual instances where this first solution is poorer). Other than for the large P1 and P2 instances, the first solutions are found within 1 seconds for all classes on average.

With the exception of the P1 instance, the final solution by GDA is far superior to both SC and SCR after 120 seconds for all instances, and in fact the final best solution is found within 30 seconds on average. For the P1 instance containing 2900 rectangles, it takes GDA over 200 seconds to find the first feasible solution (which has far better area utilization than the solutions by SC and SCR). This highlights a weakness of our approach: for large problems where the solution requires several bins, a significant amount of computation time is required both to calculate the values of U_j for the upper bound for each bin type j, and also for the ImproveSol procedure to consider all combinations of two bins.

6.3. Additional analysis

The ImproveSol post-improvement procedure takes up a major part of the computation time for our GDA. It is invoked whenever a feasible solution is found, either by the TabuPack procedure or during the computation of the lower bound for 2DVSBPP. To evaluate the effectiveness of the ImproveSol procedure, we performed an additional experiment using all the instances from PS class. We ran GDA without using the ImproveSol procedure and compared it with our original version.

The results are shown in Fig. 8, which plots the average packing utilization for these instances over time. It can be clearly seen that GDA with ImproveSol outperforms the version without over the entire 120 seconds. The effect is most pronounced early in the process, where the difference in average area utilization is over 2% at 2 seconds of computation time.

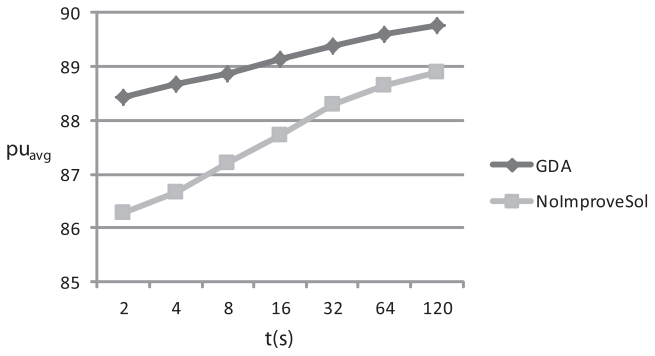


Fig. 8. Effect of ImproveSol on PS instances.

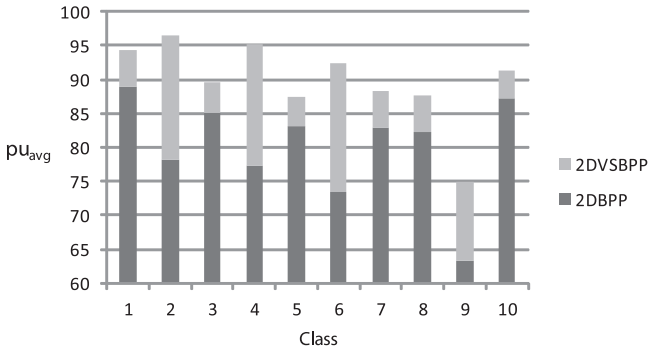


Fig. 9. Effect of variable bin sizes.

Finally, we provide some analysis on the effect of variable bin sizes on the two-dimensional bin packing problem. Fig. 9 is a stacked graph showing the difference between the average area utilization of our solutions for the 2DBPP compared to its corresponding solutions for the PS class for the 2DVSBPP (recall that the PS class was generated from the 2DBPP instances, where each instance contains an additional five types of bins). We see that the difference in average area utilization is greatest for classes 2, 4 and 6. It is no coincidence that the solutions for these classes also contain the smallest number of bins out of all the classes. These results reveal an important aspect of 2D bin packing problems: when the total number of used bins is small, the relative effect of having additional bin types increases. Therefore, the benefit of having different stock sizes is relatively larger when the total number of pieces of stock used is small.

7. Conclusions

In this paper, we proposed a goal-driven approach to the two-dimensional bin packing and variable-sized bin packing problems. The strategy behind the approach is to perform multiple binary searches on the objective values, doubling the search effort each time. Our approach includes a tabu search that uses a sequential packing heuristic, a local search that attempts to pack all unpacked rectangles in a near-feasible solution, and a post-improvement procedure to reduce the number of bins or the total area of used bins in a feasible solution. We also propose a new lower bound measure for the 2DVSBPP. Experiments on benchmark test data show our resultant goal-driven approach (GDA) outperforms all other approaches on average in terms of average solution quality for both the 2DBPP and the 2DVSBPP.

One avenue of future research is to consider some common additional requirements in cutting and packing, like the guillotine cut constraint or permitting the rotation of rectangles. These requirements can be handled simply by replacing our packing heuristic with one that fulfills these conditions, although the effectiveness of doing so must be empirically investigated. Our GDA can be potentially adapted to solve other variants of multi-dimensional bin packing problems; possibilities include three-dimensional bin packing and the variable bin cost 2DVSBPP.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.ejor.2012.08.005.

References

Berkey, J.O., Wang, P.Y., 1987. Two-dimensional finite bin-packing algorithms. The Journal of the Operational Research Society 38, 423–429.

Boschetti, M.A., Mingozzi, A., 2003a. The two-dimensional finite bin packing problem. Part I: new lower bounds for the oriented case. 4OR: A Quarterly Journal of Operations Research 1, 27–42.

Boschetti, M.A., Mingozzi, A., 2003b. The two-dimensional finite bin packing problem. Part II: new lower and upper bounds. 4OR: A Quarterly Journal of Operations Research 1, 135–147.

Burke, E.K., Kendall, G., Whitwell, G., 2004. A new placement heuristic for the orthogonal stock-cutting problem. Operations Research 52, 655–671.

Clautiaux, F., Carlier, J., Moukrim, A., 2007. A new exact method for the two-dimensional bin-packing problem with fixed orientation. Operations Research Letters 35, 357–364.

DEIS, 2012. Operations Research Group Library of Instances: Two-dimensional Bin Packing Problem. <http://www.or.deis.unibo.it/research_pages/ORinstances/2BP.html> (accessed 13.03.2012).

Dell'Amico, M., Martello, S., Vigo, D., 2002. A lower bound for the non-oriented two-dimensional bin packing problem. Discrete Applied Mathematics 118, 13–24.

Faroe, O., Pisinger, D., Zachariasen, M., 2003. Guided local search for the three-dimensional bin-packing problem. INFORMS Journal on Computing 15, 267–283.

Hopper, E., 2000. Two-dimensional Packing Utilising Evolutionary Algorithms and Other Meta-heuristic Methods. Ph.D. Thesis, University of Wales, Cardiff School of Engineering.

Hopper, E., Turton, B.C.H., 2002a. An empirical study of meta-heuristics applied to 2D rectangular bin packing – part I. Studia Informatica Universalis 2, 77–92.

Hopper, E., Turton, B.C.H., 2002b. An empirical study of meta-heuristics applied to 2D rectangular bin packing – part II. Studia Informatica Universalis 2, 93–106.

Hopper, E., Turton, B.C.H., 2002c. Problem generators for rectangular packing problems. Studia Informatica Universalis 2, 123–136.

Lodi, A., Martello, S., Monaci, M., 2002a. Two-dimensional packing problems: a survey. European Journal of Operational Research 141, 241–252.

Lodi, A., Martello, S., Vigo, D., 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS Journal on Computing 11, 345–357.

Lodi, A., Martello, S., Vigo, D., 2002b. Recent advances on two-dimensional bin packing problems. Discrete Applied Mathematics 123, 379–396.

Monaci, M., Toth, P., 2006. A set-covering-based heuristic approach for bin-packing problems. INFORMS Journal on Computing 18, 71–85.

Ortmann, F.G., Ntene, N., van Vuuren, J.H., 2010. New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. European Journal of Operational Research 203, 306–315.

Parreño, F., Alvarez-Valdes, R., Oliveira, J., Tamarit, J., 2010. A hybrid GRASP/VND algorithm for two- and three-dimensional bin packing. Annals of Operations Research 179, 203–220.

Pisinger, D., Sigurd, M., 2005. The two-dimensional bin packing problem with variable bin sizes and costs. Discrete Optimization 2, 154–167.

Pisinger, D., Sigurd, M., 2007. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. INFORMS Journal on Computing 19, 36–51.

Valenzuela, C.L., Wang, P.Y., 2001. Data set generation for rectangular placement problems. European Journal of Operational Research 134, 378–391.

Wang, P.Y., 1983. Two algorithms for constrained two-dimensional cutting stock problems. Operations Research 31, 573–586.

Wäscher, G., Haußner, H., Schumann, H., 2007. An improved typology of cutting and packing problems. European Journal of Operational Research 183, 1109–1130.

Wei, L., Oon, W.-C., Zhu, W., Lim, A., 2011. A skyline heuristic for the 2D rectangular packing and strip packing problems. European Journal of Operational Research 215, 337–346.

Wei, L., Zhu, W., 2011. Detailed Results for the Goal-driven Approach to the 2D Bin Packing and Variable-sized Bin Packing Problems. <http://www.computational-logistics.org/orlib/2dvsbpp/index.html> (accessed 13.03.2012).

Zhang, Z., Guo, S., Zhu, W., Oon, W.-C., Lim, A., 2011. Space defragmentation heuristic for 2D and 3D bin packing problems. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence.