A priority heuristic for the guillotine rectangular packing problem

Article *in* Information Processing Letters · January 2016

DOI: 10.1016/j.ipl.2015.08.008

CITATIONS
9

READS
2,032

4 authors:

Defu Zhang
Xiamen University
107 PUBLICATIONS **1,935** CITATIONS
SEE PROFILE

Leyuan Shi
University of Wisconsin–Madison
199 PUBLICATIONS **2,674** CITATIONS
SEE PROFILE

Stephen C. H. Leung
The University of Hong Kong
85 PUBLICATIONS **3,278** CITATIONS
SEE PROFILE

Tao Wu
38 PUBLICATIONS **716** CITATIONS
SEE PROFILE

Information Processing Letters 116 (2016) 15–21

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A priority heuristic for the guillotine rectangular packing problem

Defu Zhang [a,b,*], Leyuan Shi [b], Stephen C.H. Leung [c], Tao Wu [b]

[a] Department of Computer Science, Xiamen University, 361005, China
[b] Department of Industrial and Systems Engineering, University of Wisconsin-Madison, USA
[c] Department of Management Sciences, City University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 5 April 2014
Received in revised form 21 June 2015
Accepted 6 August 2015
Available online 21 August 2015
Communicated by S.M. Yiu

Keywords:
Packing problem
Heuristic algorithm
Recursive
Design of algorithms

ABSTRACT

A new priority heuristic is presented for the guillotine rectangular packing problem. This heuristic first selects one available item for a given position by a priority strategy. Then it divides the remaining space into two rectangular bins and packs them recursively, and its worst-case time complexity is $T(n) = O(n^2)$. The proposed algorithm is a general, simple and efficient method, and can solve different packing problems. Computational results on a wide range of benchmark problems have shown that the proposed algorithm outperforms existing heuristics in the literature, on average.

1. Introduction

Cutting and packing problems are related to many areas of operations research as they have diverse industrial applications such as apparel manufacturing, glass cutting, multiprocessor task scheduling, cargo loading and integrated circuit layout design. These applications can be formulated as packing problems with their respective constraints and objectives. Possible constraints include guillotine cutting and fixed orientation packing. Guillotine cutting is often required in many industrial fields since the machine cuts different types of materials into many small pieces using orthogonal cutting. The objective is to minimize the waste or height of material or maximize space utilization in the bin.

According to the typology of packing problems in Wäscher et al. [17], the guillotine rectangular packing problems (GRPP) include two classes, each class includes two variants:

- Strip packing problem (SPP): Given an open bin of width W and unlimited height, and a set of n rectangular items with sizes (h_i, w_i), $i = 1, \ldots, n$, the objective is to place each item in the bin without overlapping, such that the bin's required height is minimized. This problem includes two variants: OG and RG, where O denotes the case where items are placed with a fixed orientation, G denotes guillotine constraints are required, and R denotes items may be rotated by 90 degrees.
- Single bin packing problem (SBPP): Given a rectangular bin of width W and height H, and a set of n rectangular items, the objective is to maximize space utilization (or "filling rate") of the rectangular bin. Similarly, this problem includes two-variants: OG and RG.

GRPP is NP-hard and is difficult to solve (Belov [1], Messaoud et al. [12]). Some researches for GRPP have shown exact algorithms only solve small-scale problems (Hifi and

* Corresponding author at: Department of Computer Science, Xiamen University, 361005, China. Tel.: +86 0592 5918207; fax: +86 0592 2580258.
E-mail address: dfzhang@xmu.edu.cn (D. Zhang).

http://dx.doi.org/10.1016/j.ipl.2015.08.008
0020-0190/

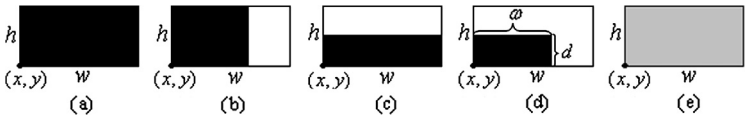

Fig. 1. Possible cases for S while placing one item.

M'Hallah [6], Cui et al. [5]), heuristic algorithms are preferred if fast computing is required for real-world applications.

Constructive heuristic algorithms cannot guarantee a solution of good quality but they can find a feasible solution in relatively short time. In particular, they can be combined with exact algorithms and metaheuristic algorithms. Coffman et al. [3], Coffman and Shor [4] presented several level-oriented heuristic algorithms: first-fit decreasing height (FFDH) algorithm, best-fit decreasing height (BFDH) algorithm. Lodi et al. [10] introduced the floor-ceiling (FC) algorithm. Martello et al. [11] also developed a heuristic algorithm (JOIN). Zhang et al. [20] presented a heuristic recursive algorithm (HR). Bortfeldt [2] further improved BFDH and obtained a BFDH* algorithm. Polyakovsky and M'Hallah [16] presented a new guillotine bottom left (GBL) heuristic algorithm. Ortmann et al. [14] developed four new and improved heuristic algorithms: modified size-alternating stack algorithm (SASm), best fit with stacking algorithm (BFS), stack ceiling {with re-sorting} algorithm (SC{R}), and stack level algorithm (SL_5).

Based on constructive heuristics, metaheuristics are widely applied to solve GRPP (Bortfeldt [2], Polyakovsky and M'Hallah [16], Wei et al. [19], Hong et al. [7]), and obtained some excellent results. Therefore, it is important to design a general heuristic algorithm that can quickly find a solution for GRPP.

2. New priority heuristic algorithm

A recursive technique is useful for GRPP as it may be used to restrict items' locations such that they can satisfy the guillotine constraint. The concept is simple as a rectangular space may be divided into several smaller rectangular spaces, and each smaller space can be divided recursively. From Fig. 1(a), we observe that the rectangular bin S is determined by its position (x, y), and its width w and its height h. The core purpose of the proposed algorithm is to fill S efficiently.

Each unplaced item can be placed into S resulting in five scenarios (the placed item is marked in black): Fig. 1(a)–(e) express cases (a)–(e) respectively. It can be observed intuitively that case (a) is the best because the item fills up the whole space. Cases (b) and (c) are better than case (d) because the remaining space in cases (b) and (c) are rectangles, while the remaining space in case (d) requires careful partitioning. Case (e) represents that no item can be placed into S and S is wasted. Which of cases (b) or (c) is better depends on the practical problems. Different problems may consider different sorting of items. For strip packing problem, sorting is usually selected by non-increasing height. Under this case, case (b) is said to be better than case (c) because the item with the larger height may have more chance to be selected to place first.

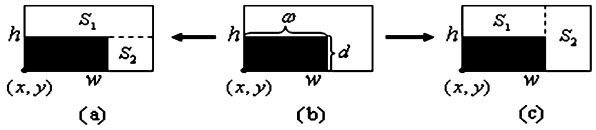

Fig. 2. The way of partitioning of S.

Similarly, if sorting is done by non-increasing width, then case (c) is better than case (b). Assume that the items are sorted by non-increasing height, then items that match cases (a), (b), (c), (d) and (e) are assigned priority 1, 2, 3, 4 and 5, respectively.

Among unplaced items, those with the highest priority (1 is the highest) are chosen for placement first. Stop packing S when case (e) occurs because all unpacked items cannot be packed into S. For cases (b) and (c), the number of bins to be placed does not increase. If two or more items have the same priority, then the first hit item in the sorted list is packed first. For case (d), the division of S is important, and is done as follows: Let $\min w$ and $\min h$ be the two parameters related to all unplaced items where, for fixed orientation packing, $\min w$ is the minimum width of all unplaced items and $\min h$ is the minimum height of all unplaced items. According to Fig. 2(b), if the value $w - \omega$ is less than $\min w$, S is divided into S_1 and S_2, as in Fig. 2(a) instead of as in Fig. 2(c), which implies that S_2 will be wasted, however, S_2 in Fig. 2(c) is larger than S_2 in Fig. 2(a). Therefore, this partition can make S_1 larger, so that it can be used by other unplaced items. Similarly, if the value $h - d$ is less than $\min h$, then S is divided into S_1 and S_2, as in Fig. 2(c), implying that S_1 will be wasted. Otherwise, there are two ways to divide S, as in Fig. 2(a) and (c). One partition is as in Fig. 2(a), another partition is shown in Fig. 2(c). Which partition is selected depends on if ω is less than $\min w$. If ω is less than $\min w$, then the former is selected because condition $\omega < \min w$ leads to waste of bin S_1 as in Fig. 2(c).

In fact, orientation of the partition is the determining factor and depends on the way the items are sorted. The aim of the strip packing problem is to minimize the height of the bin, such that the partition in Fig. 2(c) is more efficient because items with large heights have greater chance to be placed. For the single bin packing problem, orientation of the partition is mainly determined by the way the items are sorted. Orientation of the partition is shown in Fig. 2(c), when items are sorted by non-increasing width, because the case of $\omega < \min w$ rarely occurs and items with large heights have a greater chance to be placed. In fact, partitioning may be proceeded more carefully by considering $d < \min h$ or by taking other factors into account. For example, for the case of $d < \min h$, dividing S into S_1 and S_2 as in Fig. 2(a) will result the wastage of S_2. In this case, the partition as in Fig. 2(c) is better. However, it is a complex issue and depends on the nature of the

```
RecursivePacking(x, y, w, h)
  if no item can be placed into S or all the items are placed into S, then return;
  else
    for each unplaced item i do
        for j = 0 to D do determine the priority of item i;
    select an item r with highest priority from unplaced items, record its orientation j;
    if the highest priority is less than 5 then
      place item r into S by the orientation j;
      switch (priority)
        case 1: break;
        case 2: RecursivePacking(x + ω, y, w − ω, h); break;
        case 3: RecursivePacking(x, y + d, w, h − d); break;
        case 4: if w − ω < min w then
                    divide S (as Fig. 2(a)) into S1 and S2 that S2 is wasted;
                    RecursivePacking(x, y + d, w, h − d);
                else if h − d < min h then
                    divide S (as Fig. 2(c)) into S1 and S2 that S1 is wasted;
                    RecursivePacking(x + ω, y, w − ω, h);
                else if ω < min w then divide S into S1 and S2 (as in Fig. 2(a));
                    else divide S into S1 and S2 (as in Fig. 2(c));
                        Recursively pack the larger space among S1 and S2;
                        Recursively pack the smaller space among S1 and S2;
                break;
```

problem. Above is the pseudocode for the recursive packing process for S, where D denotes whether the problem considers orientation constraints or not, $D = 0$ denotes the items are placed only with the fixed orientation and $D = 1$ denotes that rotation is permitted. For item r, ω and d are width and height, respectively, if orientation value $D = 0$. Otherwise, ω and d are height and width of item r, respectively. The highest priority is an integer from 1 to 4 which corresponds to four cases from (a) to (d) in Fig. 1 respectively, so priority in the switch operator is between 1 and 4. Where the larger space (the larger area) should be filled first that is from the experience of packing by humans. More space is wasted because small items may occupy the space that can be used for large items, such that large items cannot be placed (without increasing the height).

From the above analysis, RecursivePacking(x, y, w, h) is different from HR (Zhang et al. [20]) because HR only considers the first item that can be placed into S, then divides S into two rectangular bins according to h and w. If $w < h$, then S is divided as in Fig. 2(a), otherwise it is divided as in Fig. 2(c). Since assigning priorities is the central idea behind the proposed algorithm, it is named the priority heuristic (PH) algorithm. PH is based on a recursive structure and is a general approach. Now, we use PH to solve different problems. For a strip packing problem with OG, PH is as follows:

(1) Let $D = 0$, sort all items by non-increasing height, $H = 0$, $x = 0$, $y = 0$.
(2) Place an item i by using a level structure into the open bin B, $x = w_i$, $y = H$, $w = W - w_i$, $h = h_i$, $H = H + h$, and divide the open bin into B_1 and S.
(3) RecursivePacking(x, y, w, h).
(4) If unpacked items remain, let $B = B_1$, go to (2); otherwise stop.

For the strip packing problem with RG, PH is as follows:

(1) Let $D = 1$, for each item, swap its width and height if its width is larger than its height, sort all items by non-increasing width, $H = 0$, $x = 0$, $y = 0$.
(2) If $h_i > W$ then $x = w_i$, $y = H$, $w = W - w_i$, $h = h_i$, $H = H + h$; otherwise, $x = h_i$, $y = H$, $w = W - h_i$, $h = w_i$, $H = H + h$.
(3) RecursivePacking(x, y, w, h).
(4) If unpacked items remain, let $B = B_1$, go to (2); otherwise stop.

Where H is the height of the level. H is the objective value of the problem when the algorithm stops. For single bin packing problem with OG, PH is as follows:

(1) Let $D = 0$, for each item, sort all items by non-increasing width.
(2) RecursivePacking$(0, 0, W, H)$.

For single bin packing problem with RG, PH is as follows:

(1) Let $D = 1$, for each item, swap its width and height if its width is larger than its height, sort all items by non-increasing width.
(2) RecursivePacking$(0, 0, W, H)$.

From the above descriptions of PH for different problems, observe that PH first sorts the items by non-increasing height for the strip packing problem with fixed orientation constraints (OG) because the problem is to minimize the height of the bin. Therefore, items with large heights should be placed first. PH first swaps an item's width and height if its width is larger than its height, and sorts all items by non-increasing width for strip packing problem (RG). For this problem, if $w_i > w_j$, then the probability of $h_i > h_j$ will be higher after Step (1) is finished, so large items have a greater chance to be placed first. For single bin (RG and OG) packing problems, the objective is to maximize the filling rate. For RG problem, sorting by non-increasing width makes large items have greater chance to

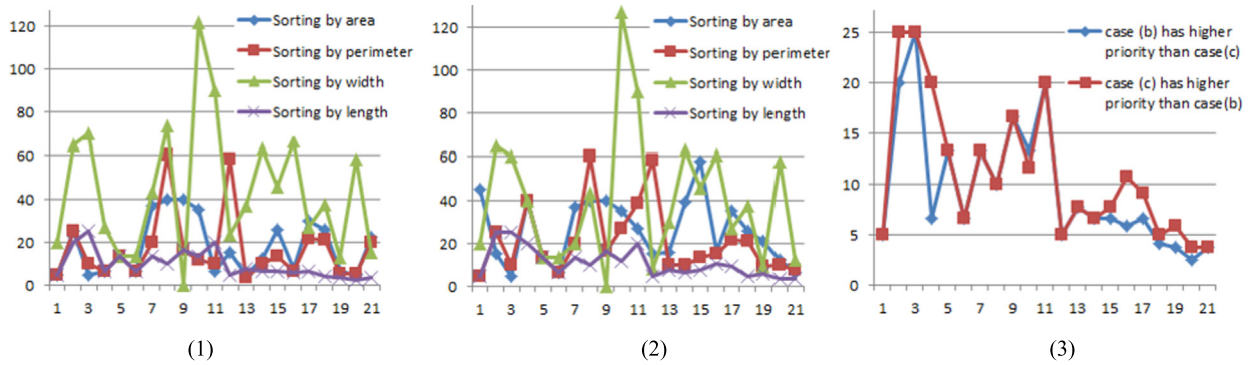

Fig. 3. Effect of some strategies.

be placed first. However, for OG problem, sorting by non-increasing width is to keep it consistent with RG problem.

It is of great interest to note that different sorting methods have different impacts on the performance of PH. We test PH by using the data set C (Hopper and Turton [8]). Fig. 3 reports results of PH for SPP with OG, where Fig. 3(1) depicts results of four sorting methods when case (b) has higher priority than case (c), and Fig. 3(2) depicts results of four sorting methods when case (c) has higher priority than case (b). It is found that sorting by height can obtain the best results. Fig. 3(3) reports results of PH with sorting by height, if case (b) has higher priority than case (c), and if case (c) has higher priority than case (b). From Fig. 3(3), it is observed that case (b) should be given higher priority than case (c) if sorting is by height. Where, X-axis in Fig. 3 denotes problem instance and Y-axis in Fig. 3 denotes Gap obtained by sorting strategies.

3. Computational complexity

PH makes use of an insertion sort algorithm, so it needs at most $O(n^2)$ time in the sorting step. According to the RecursivePacking algorithm, we select one item, going by the assigned priority, from all unplaced items, for one packing bin. We need to calculate priorities of n items, each calculation needs constant time, and the total time is $O(n)$ for the first packing. The number of unplaced items decreases as items are placed one by one. Thus, for the second packing, $O(n-1)$ time is needed. When there is only one unplaced item left, it needs $O(1)$ time. So the total running time is:

$$O(n) + O(n-1) + \cdots + O(1) = O(n^2) \quad (1)$$

It is possible that a packing bin cannot be filled by any unplaced item(s). However, one bin is divided into at the most two bins after one item is placed, and then to determine whether two bins can be filled or not can be finished in constant time, hence the worst case of PH remains $O(n^2)$. In fact, the priority heuristic algorithm (PH) requires less time because it needs only one constant time unit when packing one item to construct one layer for the strip packing problem. In addition, packing the first item with priority value 1 means PH does not need to calculate priorities of the remaining unplaced items.

4. Computational results

PH is run on a Windows XP notebook computer with a 1.73 GHz CPU and 504 MB RAM.

4.1. Strip packing problem (OG)

The priority heuristic algorithm (PH) was tested with the same instances as used by Ortmann et al. [14]. These instances were generated by Hopper and Turton [8], Wang and Valenzuela [18] and Mumford-Valenzuela et al. [13]. The optimal heights or the lower bounds of these instances are known. Algorithms compared include FC (Lodi et al. [10]), BFDH* (Bortfeldt [2]), SASm (Ortmann et al. [14]), SL_5 (Ortmann [21]) and PH.

Computational results of FC, BFDH*, and SASm are taken from Ortmann et al. [14], while SL_5 is from Ortmann [21]. These results have shown that they are the best heuristics for the OG strip and variable bin size bin packing problems. The four heuristic algorithms were executed on a Windows XP PC with a 3.0 GHz Intel Core 2 Duo CPU and 4 GB RAM. Computational results are reported in Table 1. AGap denotes the average Gap of each algorithm for a given problem class, where Gap (in %) denotes the percentage of deviation from the lower bound (LB), namely $\text{Gap} = 100 \times (\text{H-LB})/\text{LB}$. The best Gaps obtained by the five heuristic algorithms are in bold letters. AGap of PH is underlined if its AGap is the smallest.

From Table 1, we can observe that PH outperforms all 4 heuristic algorithms in terms of AGap for C1–C7, T1–T7 and Path. PH improves the current best Gap of 3 out of 7 problem instances (C1–C7), 3 out of 7 problem instances (T1–T7), 2 out of 7 problem instances (N1–N7), 2 out of 8 problem instances (Nice) and 7 out of 8 problem instances (Path).

4.2. Strip packing problem (RG)

To verify the performance of PH for SPP (RG), 480 instances from Mumford-Valenzuela et al. [13] were used. Table 2 reports the computational results, where height is the height of the open bin as obtained by different algorithms. The symbol "–" denotes that results of the corresponding algorithm are not reported. All these results are directly taken from Bortfeldt [2]. Their tests were carried

Table 1
Results of different heuristic algorithms for strip packing problem (OG).

	FC	BFDH*	SASm	SL_5	PH		FC	BFDH*	SASm	SL_5	PH
C1	**13.3**	**13.3**	28.3	15.0	16.7	C5	9.6	12.6	11.1	8.1	**7.0**
C2	**8.9**	11.1	11.1	**8.9**	**8.9**	C6	**5.6**	8.9	11.1	7.5	**5.6**
C3	17.8	18.9	16.7	14.4	**13.3**	C7	4.9	8.8	9.0	5.1	**3.3**
C4	12.8	23.3	15.6	**10.6**	12.8	AGap	10.4	13.8	14.7	10.0	9.7
T1	33.3	41.9	41.2	36.8	**31.1**	N1	**21.1**	27.1	27.1	23.9	26.6
T2	**26.1**	33.4	31.2	26.2	28.8	N2	**18.6**	21.1	30.6	18.7	19.3
T3	22.1	32.2	34.6	**21.0**	21.9	N3	19.8	30.4	28.6	20.3	**18.7**
T4	12.2	18.6	23.5	11.8	**11.5**	N4	12.8	22.2	22.5	13.5	**11.4**
T5	11.6	22.3	17.4	**11.4**	11.6	N5	11.5	24.0	18.9	**10.1**	11.3
T6	13.1	21.9	16.7	**8.6**	9.5	N6	12.3	21.1	17.7	**7.8**	9.9
T7	10.7	16.6	10.8	5.9	**5.0**	N7	9.0	16.5	14.4	5.3	**4.0**
AGap	18.4	26.7	25.1	17.4	17.06	AGap	15.0	23.2	22.8	14.2	14.46
Nice1	21.1	23.5	27.7	**21.0**	24.5	Path1	28.5	38.4	46.4	29.1	**20.1**
Nice2	16.1	18.5	23.5	16.3	**12.1**	Path2	25.4	39.5	35.8	25.1	**19.2**
Nice3	11.7	13.7	18.9	11.1	**10.8**	Path3	23.9	38.1	29.9	17.7	**6.2**
Nice4	8.9	10.1	15.6	**8.6**	9.5	Path4	23.3	34.5	18.7	10.9	**7.7**
Nice5	5.9	6.5	12.7	**5.8**	6.0	Path5	22.5	30.1	12.1	**6.3**	7.9
Nice1t	4.5	4.8	11.4	**4.3**	4.4	Path1t	23.6	30.9	10.8	5.1	**4.0**
Nice2t	3.1	3.3	10.6	**3.0**	3.3	Path2t	20.3	25.3	9.4	3.7	**3.1**
Nice5t	2.1	2.2	10.1	**2.0**	**2.0**	Path5t	20.3	24.3	7.3	2.8	**2.1**
AGap	9.2	10.3	16.3	9.0	9.1	AGap	23.5	32.6	21.3	12.6	8.8

Table 2
Results of PH, BFDH, BFDH* and GA for the strip packing problem (RG).

Instances	BFDH*	PH		SPGAL		Instances	BFDH*	PH		SPGAL	
	H	*H*	Time	*H*	Time		*H*	*H*	Time	*H*	Time
Nice1	116.2	**114.0**	0.002	105.2	103	Path1	113.6	114.2	0.001	102.8	59
Nice2	113.3	**111.9**	0.004	105.0	542	Path2	114.1	**108.1**	0.004	102.6	327
Nice3	109.1	111.7	0.005	105.3	365	Path3	112.0	**104.6**	0.005	103.1	381
Nice4	107.2	**105.1**	0.007	105.5	216	Path4	110.2	**103.9**	0.008	106.7	50
Nice5	104.6	**103.2**	0.023	103.7	496	Path5	107.5	**103.3**	0.028	105.4	93
Nice1t	103.4	**102.6**	0.048	102.7	504	Path1t	107.4	**102.3**	0.068	104.6	111
Nice2t	102.4	**101.4**	0.076	102.0	323	Path2t	105.1	**101.5**	0.165	104.0	195
Nice5t	101.7	**100.9**	0.121	101.5	305	Path5t	–	**100.2**	0.357	–	–

out on a Pentium PC with a core frequency of 2 GHz. From Table 2, we can observe that among the three heuristics, PH achieves better results for most instances except for Nice2 and Path1 and is faster than SPGAL.

4.3. Single bin packing problem (OG and RG)

To verify the performance of PH for the single bin packing problem, 21 instances from Hopper and Turton [8] were used. Polyakovsky and M'Hallah [16] reported computational results of GBL and A-B for OG and RG problems. Table 3 reports the filling rates of three algorithms. All these results are taken from Polyakovsky and M'Hallah [16]. For OG problem, we can observe that PH performs better for 13 of 21 instances and performs poorer for 8 of 21 instances, where the best results are in bold font. For RG problem, we can observe that PH performs better for 14 of 21 instances, and performs poorer for 6 of 21 instances. Compared with the complicated and advanced A-B algorithm, where the best results are in underlined, we can observe that PH performs poorer for OG problem, but performs better for RG problem on average. In particular, instance c2_2 is solved optimally by PH with 100% usage.

To verify the performance of PH in large problem instances, we compared PH with GBL and FFDH. Data set ZDF is from Leung and Zhang [9] and Zhang et al. [22] and CX is from Pinto and Oliveira [15]. These two data sets include extra large-scale instances ($n > 10\,000$). Large-scale data sets Nice1t, Nice2t, Nice5t, Path1t, Path2t and Path5t, having 10 instances each, are included. These 60 instances are non-zero-waste instances because of the transformation from float to integer data by multiplying the original data by 10 and rounding to the nearest integer. They can be downloaded from http://algorithm.xmu.edu.cn/Download.aspx#p4. Table 4 reports the filling rates of FFDH, GBL and PH. The results show that PH outperforms GBL and PH completely, except for 500 cx.

5. Conclusions

A new priority heuristic (PH) algorithm for GRPP is proposed, and four variations of the algorithm are examined. PH is a simple and very fast deterministic single-pass heuristic. Computational results have shown that PH outperforms existing heuristics from the literature for most benchmark instances. In some instances, PH improves on the best results of existing heuristics. One advantage of

Table 3
Results of PH, GBL and A-B for the single bin packing problem (OG and RG).

Instance	OG			RG		
	PH	GBL	A-B	PH	GBL	A-B
C1_1	**95.50**	88.00	92.30	**95.50**	91.50	92.30
C1_2	**86.75**	79.50	86.50	**89.00**	86.75	89.76
C1_3	**91.25**	78.50	86.00	92.50	**92.75**	89.73
C2_1	**89.00**	86.67	95.50	87.33	**87.67**	96.13
C2_2	90.50	**92.67**	98.33	**100.00**	92.67	93.48
C2_3	91.00	**94.83**	98.00	95.00	95.00	97.14
C3_1	91.06	**92.33**	81.11	**95.28**	94.67	84.20
C3_2	**86.94**	86.11	94.22	88.17	**90.00**	91.86
C3_3	87.22	**87.67**	86.00	**94.22**	91.67	93.11
C4_1	**95.25**	94.83	95.61	96.81	**97.00**	95.56
C4_2	**92.83**	92.58	95.61	**95.00**	91.19	96.60
C4_3	**93.33**	89.50	97.64	95.92	**97.22**	96.78
C5_1	**97.17**	92.80	96.28	**98.67**	95.67	96.90
C5_2	**88.72**	88.56	91.13	**97.50**	93.93	95.96
C5_3	**97.46**	94.98	90.98	**97.41**	94.85	95.62
C6_1	**96.04**	95.47	96.97	**97.88**	97.57	98.07
C6_2	92.19	**94.93**	94.25	**97.45**	96.98	96.93
C6_3	93.10	**94.98**	96.68	**96.56**	94.86	98.47
C7_1	**96.02**	95.29	97.03	**97.06**	96.07	97.03
C7_2	95.39	**97.68**	96.39	**97.84**	97.38	97.73
C7_3	93.58	**95.32**	97.56	96.70	**97.50**	98.83
Average	**92.40**	91.10	93.53	**95.32**	93.95	94.87

Table 4
Results of different algorithms for large problem instances (OG).

Instances	n	H	W	PH	FFDH	GBL
50 cx	50	600	400	**32.91**	30.45	30.72
100 cx	100	600	400	**92.13**	77.36	89.20
500 cx	500	600	400	95.13	67.41	**96.05**
1000 cx	1000	600	400	**96.34**	83.72	94.64
5000 cx	5000	600	400	**99.38**	80.01	96.30
10 000 cx	10 000	600	400	**100.00**	81.52	**100.00**
15 000 cx	15 000	600	400	**100.00**	81.16	**100.00**
Average				**87.98**	71.66	86.70
Nice1t	1000	1000	1000	**95.61**	93.97	86.93
Nice2t	2000	1000	1000	**97.05**	95.88	87.32
Nice5t	5000	1000	1000	**98.03**	97.01	87.08
Path1t	1000	1000	1000	**95.07**	75.66	91.48
Path2t	2000	1000	1000	**96.72**	75.67	92.82
Path5t	5000	1000	1000	**98.28**	81.23	92.92
Average				**96.79**	86.57	89.76
zdf1	580	330	100	**96.10**	39.07	88.99
zdf2	660	357	100	**96.44**	37.86	93.48
zdf3	740	384	100	**96.64**	37.13	93.89
zdf4	820	407	100	**96.91**	36.86	94.31
zdf5	900	434	100	**96.94**	35.85	94.50
zdf6	1532	4872	3000	**86.89**	80.88	84.00
zdf7	2432	4852	3000	**86.83**	80.80	83.93
zdf8	2532	5172	3000	**88.30**	85.93	87.38
zdf9	5032	5172	3000	**88.30**	86.32	87.38
zdf10	5064	5172	6000	**93.39**	91.36	84.93
zdf11	7564	5172	6000	**93.39**	91.29	84.93
zdf12	10 064	5172	6000	**93.39**	91.17	84.93
zdf13	15 096	5172	9000	**99.92**	94.47	84.93
zdf14	25 032	5172	3000	**88.30**	86.32	87.38
zdf15	50 032	5172	3000	**88.30**	86.32	87.38
zdf16	75 032	5172	3000	**88.30**	86.32	87.38
Average				**92.40**	71.75	88.11

PH is that it is based on a priority strategy. As shown in Table 4, the larger the problem size is, the better it may perform because it can select one item with highest priority among many items. Therefore, PH is particularly good for large-scaled problems and can be used in conjunction with meta-heuristics or exact methods. Future work may consider metaheuristic algorithms based on PH and extending PH for other packing problems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable suggestions that help improve this paper. This work was partially supported by a grant from City University of Hong Kong (Project No. 7002907) and the National Natural Science Foundation of China (Grant No. 61272003).

References

[1] G. Belov, Problems, models and algorithms in one- and two-dimensional cutting, Ph.D. Thesis, Technischen Universität, Dresden, 2003.
[2] A. Bortfeldt, A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces, Eur. J. Oper. Res. 172 (3) (2006) 814–837.
[3] E.G. Coffman, D.S. Garey, R.E. Tarjan, Performance bounds for level oriented two-dimensional packing algorithms, SIAM J. Comput. 9 (4) (1980) 808–826.
[4] E.G. Coffman, P.W. Shor, Average-case analysis of cutting and packing in two dimensions, Eur. J. Oper. Res. 44 (2) (1990) 134–144.
[5] Y. Cui, Y. Yang, X. Cheng, P. Song, A recursive branch-and-bound algorithm for the rectangular guillotine strip packing problem, Comput. Oper. Res. 35 (4) (2008) 1281–1291.
[6] M. Hifi, R. M'Hallah, An exact algorithm for constrained two-dimensional two-staged cutting problems, Oper. Res. 53 (1) (2005) 140–150.
[7] S. Hong, D. Zhang, C.H. Lau, X.X. Zeng, Y. Si, A hybrid heuristic algorithm for the 2D variable-sized bin packing problem, Eur. J. Oper. Res. 238 (1) (2014) 95–103.
[8] E. Hopper, B.C.H. Turton, An empirical investigation of metaheuristic and heuristic algorithms for a 2D packing problem, Eur. J. Oper. Res. 128 (1) (2001) 34–57.
[9] S. Leung, D. Zhang, A fast layer-based heuristic for non-guillotine strip packing, Expert Syst. Appl. 38 (2011) 13032–13042.
[10] A. Lodi, S. Martello, D. Vigo, Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems, INFORMS J. Comput. 11 (4) (1999) 345–357.
[11] S. Martello, M. Monaci, D. Vigo, An exact approach to the strip-packing problem, INFORMS J. Comput. 15 (3) (2003) 310–319.
[12] S.B. Messaoud, C. Chu, M.L. Espinouse, Characterization and modelling of guillotine constraints, Eur. J. Oper. Res. 191 (2008) 112–126.
[13] C.L. Mumford-Valenzuela, J. Vick, P.Y. Wang, Heuristics for large strip packing problems with guillotine patterns: an empirical study, in: Metaheuristics: Computer Decision-Making, Kluwer Academic Publishers B.V., 2003, pp. 501–522.
[14] F.G. Ortmann, N. Ntene, J.H. Van Vuuren, New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems, Eur. J. Oper. Res. 203 (2) (2010) 306–315.

[15] E. Pinto, J.F. Oliveira, Algorithm based on graphs for the non-guillotinable two-dimensional packing problem, in: Second ESICUP Meeting, Southampton, 2005.

[16] S. Polyakovsky, R. M'Hallah, An agent-based approach to the two-dimensional guillotine bin packing problem, Eur. J. Oper. Res. 192 (2009) 767–781.

[17] G. Wäscher, H. Haußner, H. Schumann, An improved typology of cutting and packing problems, Eur. J. Oper. Res. 183 (3) (2007) 1109–1130.

[18] P.Y. Wang, C.L. Valenzuela, Data set generation for rectangular placement problems, Eur. J. Oper. Res. 134 (2) (2001) 378–391.

[19] L. Wei, T. Tian, W. Zhu, A. Lim, A block-based layer building approach for the 2D guillotine strip packing problem, Eur. J. Oper. Res. 239 (1) (2014) 58–69.

[20] D. Zhang, Y. Kang, A. Deng, A new heuristic recursive algorithm for the strip rectangular packing problem, Comput. Oper. Res. 33 (8) (2006) 2209–2217.

[21] F.G. Ortmann, Heuristics for offline rectangular packing problems, Ph.D. Thesis, Stellenbosch University, 2010.

[22] D. Zhang, L. Wei, S. Leung, Q. Chen, A binary search heuristic algorithm based on randomized local search for the rectangular strip packing problem, INFORMS J. Comput. 25 (2) (2013) 332–345.