An improved priority heuristic for the fixed guillotine rectangular packing problem

Zhengyang Shang\textsuperscript{1*}, Mingming Pan\textsuperscript{2} and Jiabao Pan\textsuperscript{1}

\textsuperscript{1}School of Mechanical and Automotive Engineering, Anhui Polytechnic University, Wuhu 241000, China.
\textsuperscript{2}School of Management Engineering, Anhui Polytechnic University, Wuhu 241000, China.
\textsuperscript{*}Corresponding author’s e-mail: shangzy@ahpu.edu.cn

Abstract. An improved priority heuristic (IPH) is presented for the guillotine rectangular packing problem with a fixed orientation constraint. This algorithm can continuously divide the remaining space into blocks and recursively layout the entire space by filling each current block. IPH inherits the placement method of PH and adopts an improved space partitioning rule. The partitioning rule comprehensively considers the effects of the newly generated block on subsequent item placement. Thus, the remaining space can be filled as much as possible from a probability perspective. Given the introduction and utilization of different solving parameters, the worst-case time complexity is $O(4n^3)$. Computational results on benchmark datasets show that IPH outperforms existing heuristics on average. For large-scale instances, several perfect layouts can be obtained for the first time.

1. Introduction

The rectangular packing problem (RPP) is a basic operational problem, in which a given set of small rectangles with different sizes (abbreviated as items) is packed into a large rectangle (abbreviated as bin), so as to maximize the filling rate of a single bin or minimize the height of a strip bin. The items must be packed orthogonally without overlapping. When the guillotine constraint is considered, each cut (namely each bound among items) divides the stock block into two completely separate small blocks, this problem is called the guillotine rectangular packing problem (GRPP). Given that a machine cuts different types of materials into many small pieces via orthogonal cutting, GRPP is widely applied in industrial fields, such as glass cutting, woodcutting, and cargo loading[1].

In consideration of item orientation, GRPP generally includes two variants of OG and RG, where O denotes that the items are placed with a fixed orientation, R indicates that the items may be rotated by $90^\circ$, and G means that the guillotine constraint is required. Depending on optimization objectives, GRPP can be defined as a single bin packing problem (SBPP), which aims to maximize the space utilization of a single bin with a predefined size, and a strip packing problem (SPP), which strives to minimize the height of a bin with a given width[2]. This study focuses on rapidly solving GRPP with OG for the two objectives.

The goal of RPPs is bin packing, and the corresponding placement method is the core of all algorithms. Therefore, extensive effort has been exerted for efficient item packing. The bottom-left (BL)[3] and bottom-left-fill (BLF)[4] placement methods were proposed and usually combined with different searching methods in meta-heuristic algorithms, such simulated annealing[5], tabu[6], genetic
algorithm[7], artificial neural networks[8], and greedy randomized adaptive search procedure[9]. Despite providing good solutions, these search-based algorithms are limited by their huge computational costs. Later on, Burke, Kendall, and Whitwell[10] proposed a best-fit heuristic that can actively select a suitable-sized item to pack. Leung and Zhang[11] and Leung, Zhang, and Sim[12] introduced a scoring rule to measure the fitness of each item and classified five cases. Wei et al.[13] further improved the scoring rules and obtained the best results for most instances. To improve algorithmic efficiency, Imahori and Yagiura[14] demonstrated an efficient implementation with time complexity of $O(n \log n)$. Zhang et al.[15] presented a binary search heuristic for randomized local search. Wei et al.[16] developed a rapid implementation method with a randomized local search for solving SPP and proved that the solution is superior and faster.

Unlike in basic RPP, items cannot be randomly placed in GRPP. Thus, a recursive divide-and-conquer strategy is commonly used to meet the guillotine constraint. The key operation in this strategy is the spatial partitioning rule. Cui et al.[17] divided the placement space into segments and blocks by adopting vertical and horizontal partitioning rules. Lodi, Monaci, and Pietroboni[18] presented a single segmentation criterion and simplified the algorithmic searching method. Moreover, Lodi, Martello, and Vigo[19] introduced the floor–ceiling (FC) algorithm with tabu search, and Ortmann, Ntene, and Vuuren[20] developed the stack level algorithm (SL5). Wei et al.[21] proposed a block-based algorithm by packaging a block of items instead of a single item in each step. These algorithms introduce an iteration of optimal searching into the construction, which are time-consuming and only applicable to small-scale problems.

To rapidly solve large-scale problems, many direct solution heuristics are proposed. Coffman, Garey, and Tarjan[22] and Coffman and Shor[23] presented several level-oriented heuristics, including the first-fit decreasing height (FFDH) algorithm and the best-fit decreasing height (BFDH) algorithm. Bortfeldt[7] further improved BFDH into BFDH*. Recently, Polyakovskiy and M’Hallah[24] presented a new guillotine bottom-left (GBL) heuristic and a pseudo-parallel agent-based implementation. With the priority heuristic selection method and condition-based spatial partitioning rule, Zhang et al.[1] rapidly solved large-scale GRPPs and achieved the best effect to date.

For the efficient solution of large-scale GRPPs, we adopt the placement method of the priority heuristic (PH) algorithm[1] and propose a new spatial partitioning rule. The improved priority heuristic (IPH) can perform effective direct packing in a probabilistic manner, which is beneficial for processing numerous items. The rest of the paper is organized as follows. Section 2 presents the new approach from four aspects, namely, placement method, spatial partitioning rule, implementation, and time complexity. Section 3 shows the solution results of IPH targeting different optimization objectives of SBPP and SPP. Section 4 provides conclusions.

2. IPH

Given the guillotine constraint, the proposed algorithm adopts the divide-and-conquer strategy to solve GRPP. Specifically, after an item is placed in a bin, the residual space is divided into two small rectangular bins (called blocks hereafter). The entire bin can be recursively filled by continuously filling each newly-formed block. A simple instance of guillotine packing is illustrated in Figure 1, where the dotted line is the guillotine divider, $b_i$ is the block being filled, and each item is always placed with its bottom-left corner in the bottom-left corner of the current block. The key factors in this process are the placement method, which means how to pack an appropriate item into the current block, and the spatial partitioning rule, which aims to obtain the generated blocks for the placement of the remaining item. From these two aspects, we elaborate on the construction, implementation pattern, and time complexity of IPH.
2.1. Placement method

To optimize the layout of the entire bin, we recursively select a block and fill it with the appropriate item. Based on the efficient placement of items, IPH inherits the priority strategy of the PH algorithm. For simplification, the rectangular block is determined by its position \((x, y)\), width \(W\), and height \(H\); the width and height of item are marked as \(w\) and \(h\), respectively. Figure 2 shows that each unplaced item can be placed in a block resulting in five scenarios. Case (a) is the best because the item fills up the entire block. In Case (e), no item can be placed in the block, and the block is wasted. From Cases (a) to (e), the packing effect is gradually worsened, as described in the PH algorithm. Thus, items that match Cases (a), (b), (c), (d) and (e) are assigned with priority 4, 3, 2, 1 and 0, respectively. A higher priority means the item should be selected and placed earlier. For items with the same priority, we prefer the higher item.

![Figure 1. Example of guillotine packing](image)

![Figure 2. Possible causes for the block while placing one item](image)

**RecursivePacking** \((x, y, W, H)\)

If no item can be placed into block or all the items are placed into block, then return;

Else

Select an item with the highest priority from unplaced items for the current block

If the highest priority is more than 0 then

Switch (priority)

Case 4: break

Case 3: RecursivePacking \((x + w, y, W - w, H)\); break;

Case 2: RecursivePacking \((x, y + d, W, H - d)\); break;

Case 1: Divide current block into \(SB_1\) and \(SB_2\) by *guillotine partitioning rule*;

  Recursively pack the smaller space among \(SB_1\) and \(SB_2\);

  Recursively pack the larger space among \(SB_1\) and \(SB_2\);

End

End
As stated in the pseudo-code `RecursivePacking(x, y, W, H)`, when the block is to be filled, the item with the higher priority will be selected and placed preferentially. If the priority of an item is 4, no residual space will be left in the current block, and the program will automatically execute for the next block. If the priority is 3 or 2, the newly generated block will be recursively packed, as shown in Figs. 2(b) and 2(c). If the priority is 0, the current block will be discarded because no remaining item can be placed in. When the item is placed with priority 1 as shown in Figure 2(d), the current block may be divided horizontally into $HB_1$ and $HB_2$ (Figure 3(a)) or vertically into $VB_1$ and $VB_2$ (Figure 3(b)). The specific block division method should be determined according to the spatial partitioning rule, and the smaller block will be recursively packed first. In terms of the placement method, IPH prioritizes filling smaller blocks, which is different from the PH algorithm.

![Figure 3. Way of block partitioning](image)

2.2. Spatial partitioning rule

When an item is assigned priority 1, a spatial partitioning rule is needed to divide the residual space into two small rectangular blocks. The partitioning rule considerably affects the final packing effect because the newly generated blocks directly determine the placement of the subsequent items.

![Figure 4. Example of layer-based heuristic packing](image)

Given that the bricklaying heuristic can be effectively used in the packing problem, we refer to and modify this method in our partitioning rule. The bricklaying heuristic continuously generates layers or skylines to guide the successive placement of items. As shown in Figure 4, the higher item is placed preferentially, and the corresponding layer is generated, which largely increases the probability for placement of subsequent higher items. Similarly, this recursive layer-based heuristic can be introduced to the division of residual space. However, under the guillotine constraint, the blocks are independent, and the wasted space cannot be re-integrated and utilized. Simple layer-based strategies often result in wasted space due to the non-consideration of the shape characteristics of subsequent items. Thus, an improved bricklaying method based on heuristic spatial partition is proposed. Original bricklaying and improved bricklaying are compared in Figs. 5(a) and 5(b) to illustrate the operational mechanism of this partition method. The new block $B_1^*$ is larger than $B_1$, which contributes to the placement of the subsequent larger items and may further improve the entire bin filling effect. While the new block $B_2^*$ is smaller than $B_2$, less unusable space is likely to be created. In other words, when a block is divided into two sub-blocks (the
large newly generated one is called LB, and the small one is called SB), the larger LB is more favorable for the subsequent item packing, and the smaller SB is likely to cause less wasted space. On the basis of this clue and with Figure 3 as an example, when the space $VB_1$ is sufficiently small relative to $HB_1$, IPH adopts vertical division to produce a larger space $VB_1$ and a smaller space $VB_2$, which maximizes the available space and minimizes the possible wasted space. Other situations are horizontally divided according to the layer-based rule.

![Comparison of different partitioning rules](image)

**Figure 5.** Comparison of different partitioning rules

**Guillotine partitioning rule**

If $W - w < \min w$ then

Divide Block into $HB_1$ and $HB_2$ as shown in Figure 3(a); **break**;

Else if $H - h < \min h$ then

Divide Block into $VB_1$ and $VB_2$ as shown in Figure 3(c); **break**;

Else

If $\frac{\text{area}(HB_1)}{\text{area}(VB_1)} \geq m$

Divide Block into $VB_1$ and $VB_2$ as shown in Figure 3(c); **break**;

Else

Divide Block into $HB_1$ and $HB_2$ as shown in Figure 3(a); **break**;

End

End

The partitioning rule is shown as pseudo-code **guillotine partitioning rule**, and a relevant example is illustrated in Figure 3. Let $\min w$ be the minimum width and $\min h$ be the minimum height of all unplaced items. When an item is placed in the block, if the width of the right-side residual space is narrower than that of the smallest unplaced item, or namely the right-side space will not be placed with any unplaced item ($W - w < \min w$), horizontal division is conducted to enlarge the area of usable space $HB$, and reduce the area of wasted space $HB_1$. Similarly, the vertical division is adopted in the case of $H - h < \min h$. When both the upper and right residual spaces can be utilized, the improved bricklaying method is used for block partitioning. Then an area ratio parameter $m$ is introduced to determine the choice of horizontal or vertical division. Specifically, the block will be vertically divided if $\frac{\text{area}(HB_1)}{\text{area}(VB_1)} \geq m$; otherwise, it will be divided horizontally.

2.3. Implementation

A greedy-based search method is used to rapidly determine the item with the highest priority. For the current block, each candidate item is assigned a priority according to its matching relationship with this block in terms of size. When the height and width of an item are the same as the block, its priority is 4; when its height the same as the block but its width is smaller, its priority is 3. The three other cases of priority assignment are described in Section 2.1. Afterward, the set of items is sorted according to priority in a non-increasing order (items with the same priority are arranged according to
their heights as in the bricklaying strategy). Thus, instead of checking each item one by one, we only need to find the first item in this sequence. IPH is then constructed as follows:

**IPH algorithm**

For \( m = 1 : m_{\text{max}} \)

While a block can be filled by the remaining item

Select an item with the highest priority for the current block

Write the filling rate of the bin

If the filling rate is 100%

Break

End

Output the optimal solution

Under specific conditions, parameter \( m \) decides whether the block is horizontally or vertically divided. Since \( m \) is affected by the overall dimensional characteristics of subsequent items, it is hard to determine a unified and concrete value. For this reason and given the basic idea of bricklaying, we empirically set the value of \( m \) from 1 to 20. It means that IPH divides a block horizontally to leave as much space as possible on the top and divides vertically only when the right-side space is sufficiently large. Therefore, the operation is conducted with different values of \( m \), and the optimal result is selected as the output.

2.4. Time complexity

IPH repeatedly selects the smallest block and fills it until all items are placed in or no item can be placed in. The maximum number of packing operations is equal to the total number of blocks generated. When an item with priority 4 or 0 is placed in the block, this block will be filled up or abandoned, and the bin contains one less block. When the item with priority 3 or 2 is placed in the block, a new block replaces the previous block. When the item with priority 1 is placed, the current block is divided into two small blocks. Thus, the worst situation is that all blocks are filled by items with priority 1, and the first generated block is always unavailable. So the running time can be estimated as \( O(n) + O(n) + O(n-1) + O(n-1) + \cdots + O(1) = O(2n^2) \). Owing to the introduction of parameter \( m \) with different assignments, the worst time complexity of IPH is \( O(m_{\text{max}}) \times O(2n^2) \) or \( O(40n^2) \).

3. Results and analyses

The original IPH was designed to solve SBPP under the OG constraint, but it can also be applied to SPP by gradually increasing the bin height by \( H = H + 1 \) until all items are completely packed in. For SBPP, we aim to maximize the space utilization rate of the bin, but for SPP, we minimize the gap between solving height \( H \) and lower bound \( (LB) \), namely \( \min (Gap) = \min (100 \times (H - LB) / LB) \). Several classic instances for comparison and validation are listed in Table 1, where \#inst. is the number of instances in a dataset and \( n \) is the number of items included in an instance. The best solutions obtained by all given heuristics are in bold letters. Datasets CX and ZDF include extra-large-scale instances with \( n > 10,000 \). All data are tested on a personal computer with a 1.6GHz CPU and 4 GB of RAM.

| Data source | Data set | #inst. | N    |
|-------------|----------|--------|------|
| [26]        | C        | 21     | 16-197 |
|             | NT(n)    | 35     | 17-199 |
| [27]        | NT(t)    | 35     | 17-199 |
The performance of IPH is evaluated from two aspects, including running time and packing quality. First, IPH can rapidly and directly find a solution with the worst-case time complexity $O(40^n)$. One instance is used to estimate the solving time of different algorithms on dataset CX (Figure 6). The average running time of GBL is 42.6s, while PH has an average running time of 0.12s with time complexity $O(n)$. The average time spent on IPH is 0.22s, and the running times for instances 1000cx, 5000cx, 10000cx, and 15000cx are all relatively short because any of them can quickly achieve a 100% filling effect. The existing fast algorithms, especially PH and IPH, exhibit satisfactory efficiency due to the low time complexity and efficient implementation, which are meaningful for rapidly solving large-scale instances. A comparison experiment for IPH packing performance is performed in Section 3.1 and 3.2.

![Figure 6. Computation time in GBL, PH, and IPH for dataset CX](image)

### Table 2. Results of different algorithms on dataset C (OG)

| Instance | GBL | PH | IPH | Instance | GBL | PH | IPH |
|----------|-----|----|-----|----------|-----|----|-----|
| C1-1     | 88.00 | **95.50** | 90.00 | C4-3     | 89.50 | 93.33 | **98.42** |
| C1-2     | 79.50 | 86.75 | **88.75** | C5-1     | 92.80 | 97.17 | **97.56** |
| C1-3     | 78.50 | 91.25 | **100.00** | C5-2     | 88.56 | 88.72 | **94.74** |
| C2-1     | 86.67 | 89.00 | **90.33** | C5-3     | 94.98 | 97.46 | **97.54** |
| C2-2     | 92.67 | 90.50 | **97.83** | C6-1     | 95.47 | 96.04 | **97.79** |
| C2-3     | 94.83 | 91.00 | **98.00** | C6-2     | 94.93 | 92.19 | **97.75** |
Table 3. Results of different algorithms on large-scale instances (OG)

| Instances | n   | H  | W  | GBL | PH  | IPH  | Average |
|-----------|-----|----|----|-----|-----|------|---------|
| 50cx      | 50  | 600| 400| 30.72 | 32.91 | 93.36 |
| 100cx     | 100 | 600| 400| 89.20 | 92.13 | 93.09 |
| 500cx     | 500 | 600| 400| 96.05 | 95.13 | 99.94 |
| 1000cx    | 1000| 600| 400| 94.64 | 96.34 | 100.00 |
| 5000cx    | 5000| 600| 400| 96.30 | 99.38 | 100.00 |
| 10000cx   | 10000| 600| 400| 100.00 | 100.00 | 100.00 |
| 15000cx   | 15000| 600| 400| 100.00 | 100.00 | 100.00 |
| Average   | --- | --- | --- | 86.70 | 87.98 | 98.06 |

Nice1t 1000 | 1000 | 1000 | 1000 | 86.93 | 95.61 | 93.78 |
Nice2t 2000 | 1000 | 1000 | 1000 | 87.32 | 97.05 | 95.77 |
Nice5t 5000 | 1000 | 1000 | 1000 | 87.08 | 98.03 | 97.64 |
Path1t 1000 | 1000 | 1000 | 1000 | 91.48 | 96.72 | 97.43 |
Path2t 2000 | 1000 | 1000 | 1000 | 92.82 | 98.28 | 98.22 |
Path5t 5000 | 1000 | 1000 | 1000 | 92.92 | 98.28 | 98.22 |
Average   | --- | --- | --- | 89.76 | 96.79 | 96.55 |

zdf1 580 | 330 | 100 | 88.99 | 96.10 | 99.51 |
zdf2 660 | 357 | 100 | 93.48 | 96.44 | 99.49 |
zdf3 740 | 384 | 100 | 93.89 | 96.64 | 99.47 |
zdf4 820 | 407 | 100 | 94.31 | 96.91 | 99.60 |
zdf5 900 | 434 | 100 | 94.50 | 96.94 | 99.78 |
zdf6 1532 | 4872 | 3000 | 84.00 | 86.89 | 92.20 |
zdf7 2432 | 4852 | 3000 | 83.93 | 86.83 | 92.17 |
zdf8 2532 | 5172 | 3000 | 87.38 | 88.30 | 96.55 |
zdf9 5032 | 5172 | 3000 | 87.38 | 88.30 | 96.55 |
zdf10 5064 | 5172 | 6000 | 84.93 | 93.39 | 100.00* |
zdf11 7564 | 5172 | 6000 | 84.93 | 93.39 | 100.00* |
zdf12 10064 | 5172 | 6000 | 84.93 | 93.39 | 100.00* |
zdf13 15096 | 5172 | 9000 | 84.93 | 99.92 | 100.00* |
3.2. Strip packing problem

We test the same datasets[20] to verify the performance of IPH in SPP. The optimal heights or the lower bounds of the 151 instances are known. AGap denotes the average gap of each algorithm for a given problem class. The computational results of BFDH*[7], FC[19], SL5[20], PH[1], and IPH are listed in Table 4. IPH outperforms the four other algorithms in terms of AGap for C, NT(t), NT(n), and Path. It improves the current best gap of all instances in C, five out of seven instances in NT(t), three out of seven instances in NT(n), and of all instances in Path. Although the results in Nicet are not as good, the overall performance of IPH remains satisfactory.

### Table 4. Results of different algorithms for the strip packing problem (OG)

|        | BFDH* | FC   | SL5  | PH  | IPH  | BFDH* | FC   | SL5  | PH  | IPH  |
|--------|-------|------|------|-----|------|-------|------|------|-----|------|
| C1     | 13.3  | 13.3 | 15.0 | 16.7| 5    | C5    | 12.6 | 9.6  | 8.1 | 7.0  |
| C2     | 11.1  | 8.9  | 8.9  | 8.9 | 8.8  | C6    | 8.9  | 5.6  | 7.5 | 5.6  |
| C3     | 18.9  | 17.8 | 14.4 | 13.3| 13.3 | C7    | 8.8  | 4.9  | 5.1 | 3.3  |
| C4     | 23.3  | 12.8 | 10.6 | 12.8| 6.1  | AGap  | 13.8 | 10.4 | 9.9 | 9.7  |
| NT(t1) | 41.9  | 33.3 | 36.8 | 31.1| 22.3 | NT(n1)| 27.1 | 21.1 | 23.9| 26.6 |
| NT(t2) | 33.4  | 26.1 | 26.2 | 28.8| 21.4 | NT(n2)| 21.1 | 18.6 | 18.7| 19.3 |
| NT(t3) | 32.2  | 22.1 | 21.0 | 21.9| 21.9 | NT(n3)| 30.4 | 19.8 | 20.3| 18.7 |
| NT(t4) | 18.6  | 12.2 | 11.8 | 11.5| 12.6 | NT(n4)| 22.2 | 12.8 | 13.5| 11.4 |

*: all items are placed in the bin for the non-zero-waste instance

![Figure 7](image-url). Optimal layouts of instances 1000cx and zdf10
4. Conclusions
An improved priority heuristic (IPH) for GRPP with OG constraint is proposed, which is a simple and very fast deterministic single-pass heuristic. Given the guillotine constraint, a divide-and-conquer solving strategy is used, where the block is generated continuously and filled recursively. This overall solution idea has two key elements: the placement method and spatial partitioning rule. For the placement method, IPH inherits the priority placement strategy from PH. The difference is that IPH preferentially selects and fills the smaller block, which has a fundamental impact on the operation and construction of the entire algorithm. For the spatial division, an improved partitioning rule based on the bricklaying method is proposed, which can adopt different partitioning ways according to different spatial patterns. The rule comprehensively considers the effects of the newly generated block on subsequent items placement. Thus, the remaining space can be filled as much as possible from a probability perspective. This method is completely different from the spatial partitioning strategy in previous literature. Computational results show that IPH outperforms existing heuristics for most benchmark instances. The proposed approach is effective for large-scale GRPP, and even several optimal layouts have been found for the first time. The future work is to extend IPH to other variants of the packing problem.

Data Availability
Previously reported datasets were used to support this study. These prior studies and datasets are cited at relevant places within the text as references.

Conflicts of Interest
The authors declare that they have no conflicts of interest.

Acknowledgments
This work was supported by the Natural Foundation Key Program in the Universities of Anhui Province, China (KJ2019A0148).

References
[1] Zhang, D., Shi, L., Leung, S. C. H., and Wu, T. (2016) A priority heuristic for the guillotine rectangular packing problem. Information Processing Letters, 116(1): 15-21.
[2] Wäscher, G., Häußner, H., and Schumann, H. (2007) An improved typology of cutting and packing problems. European Journal of Operational Research, 183(3): 1109-1130.
[3] Baker, B. S., Coffman, E. G. J., and Rivest, R. L. (1980) Orthogonal packing in two dimensions. SIAM Journal on Computing, 9(4): 846-855.
[4] Chazelle, B. (1983) The bottom-left bin-packing heuristic: an efficient implementation. IEEE Transactions on Computers, 32(8): 697-707.
[5] Burke, E. K., Kendall, G., and Whitwell, G. (2009) A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock-cutting problem. Informs Journal on Computing, 21(3): 505-516.

[6] Alvarez-Valdes, R., Parreño, F., and Tamarit, J. M. (2007) A tabu search algorithm for two-dimensional non-guillotine cutting problems. European Journal of Operational Research, 183(3): 1167-1182.

[7] Bortfeldt, A. (2009) A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. European Journal of Operational Research, 16(6): 814-837.

[8] Dagli, C. H., and Poshyanonda, P. (1997) New approaches to nesting rectangular patterns. Journal of Intelligent Manufacturing, 8(3): 177-190.

[9] Alvarez-Valdes, R., Parreño, F., and Tamarit, J. M. (2005) A grasp algorithm for constrained two-dimensional non-guillotine cutting problems. Journal of the Operational Research Society, 56(4): 414-425.

[10] Burke, E.K., Kendall, G., and Whitwell, G. (2004) A new placement heuristic for the orthogonal stock-cutting problem. Operation Research, 52(4): 655-671.

[11] Leung, S. C. H., and Zhang, D. (2011) A fast layer-based heuristic for non-guillotine strip packing. Expert Systems with Applications, 38(10): 13032-13042.

[12] Leung, S. C. H., Zhang, D., and Sim, K. M. (2011) A two-stage intelligent search algorithm for the two-dimensional strip packing problem. European Journal of Operational Research, 215(1): 57-69.

[13] Wei, L., Qin, H., Cheang, B., and Xu, X. (2016) An efficient intelligent search algorithm for the two-dimensional rectangular strip packing problem. International Transactions in Operational Research, 23(1-2): 65-92.

[14] Imahori, S., and Yagiura, M. (2010) The best-fit heuristic for the rectangular strip packing problem: efficient implementation and the worst-case approximation ratio. Computers and Operations Research, 37(2): 325-333.

[15] Zhang, D., Wei, L., Leung, S. C. H., and Chen, Q. (2013) A binary search heuristic algorithm based on randomized local search for the rectangular strip-packing problem. Informs Journal on Computing, 25(2): 332-345.

[16] Wei, L., Hu, Q., Leung, S. C. H., and Zhang, N. (2017) An improved skyline based heuristic for the 2D strip packing problem and its efficient implementation. Computers and Operations Research, 80: 113-127.

[17] Cui, Y., Yang, Y., Cheng, X., and Song, P. (2008) A recursive branch-and-bound algorithm for the rectangular guillotine strip packing problem. Computers and Operations Research, 35(4): 1281-1291.

[18] Lodi, A., Monaci, M., and Pietrobuoni, E. (2015) Partial enumeration algorithms for the Two-Dimensional Bin Packing problem with guillotine constraints. Discrete Applied Mathematics, 217: 40-47.

[19] Lodi, A., Martello, S., and Vigo, D. (1999) Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. Informs Journal on Computing, 11(4): 345-357.

[20] Ortmann, F. G., Ntene, N., and Vuuren, J. H. V. (2010) New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. European Journal of Operational Research, 203(2): 306-315.

[21] Wei, L., Tian, T., Zhu, W., and Lim, A. (2014) A block-based layer building approach for the 2D guillotine strip packing problem. European Journal of Operational Research, 239(1): 58-69.

[22] Coffman, E. G., Garey, D. S., and Tarjan, R. E. (1980) Performance bounds for level oriented two-dimensional packing algorithms. SIAM Journal on Computing, 9(4): 808-826.

[23] Coffman, E. G., and Shor, P. W. (1990) Average-case analysis of cutting and packing in two dimensions. European Journal of Operational Research, 44(2): 134-144.

[24] Polyakovskiy, S., and M’Hallah, R. (2009) An agent-based approach to the two-dimensional guillotine bin packing problem. European Journal of Operational Research, 192(3): 767-781.
[25] Zhang, D., Han, S., and Ye, W. (2008) A Bricklaying Heuristic Algorithm for the Orthogonal Rectangular Packing Problem. Chinese Journal of Computers, 31(3): 509-515.

[26] Hopper, E., and Turton, B. C. H. (2001) An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. European Journal of Operational Research, 128(1): 34-57.

[27] Hopper, E. (2000) Two-dimensional packing utilizing evolutionary algorithms and other meta-heuristic methods. University of Wales Cardiff, The Cardiff.

[28] Mumford-Valenzuela, C. L., Vick, J., and Wang P. Y. (2003) Heuristics for large strip packing problems with guillotine patterns: an empirical study. Metaheuristics: Computer Decision-Making. Springer US.

[29] Pinto, E., and Oliveira, J. F. (2005) Algorithm based on graphs for the non-guillotinable two-dimensional packing problem. Second ESICUP Meeting, Southampton.