Vol.2012-AL-141 No.6
2012/10/4

レクトリニア多角形配置問題に対する高速な構築型解法

胡 艶楠[1,a) 橋本 英樹[1,b) 今堀 慎治[2,c) 柳浦 睦憲[1,d)

Abstract: The rectilinear block packing problem is a problem of packing a set of rectilinear blocks into a larger rectangular container, where a rectilinear block is a polygonal block whose interior angle is either 90° or 270°. There exist many applications of this problem. In this paper, we propose a new construction heuristic algorithm based on the bottom-left strategy. The proposed algorithm is tested on a series of instances, which are generated from nine benchmark instances. The computational results show that the proposed algorithm is especially effective for large instances of the rectilinear block packing problem.

1. Introduction

The rectilinear block packing problem is a problem of packing a set of arbitrary shaped rectilinear blocks into a larger rectangular container without overlap so as to minimize or maximize a given objective function. A rectilinear block is a polygonal block, whose interior angle is either 90° or 270°. This problem involves many industrial applications, such as VLSI design, timber/glass cutting, and newspaper layout. It is among classical packing problems and is known to be NP-hard [1].

A special case of the rectilinear block packing problem is the rectangle packing problem. Up to now, many efficient algorithms have been proposed to solve the rectangle packing problem, such as simulated annealing [2], hybrid algorithm [3] and quasi-human heuristic algorithm [4]. The *bottom-left algorithm* [1] and the *best-fit algorithm* [5] are known as the most remarkable ones among the typical frameworks of existing construction heuristic algorithms. Inspired by these approaches, we proposed a bottom-left algorithm and a best-fit algorithm for the rectilinear block packing problem, and we also designed their efficient implementations in [6].

The main strategy of our algorithms is the *bottom-left strategy*, which derives from the bottom-left algorithm for rectangle packing [1]. In this strategy, whenever a new item is being packed into the container, it will be placed at the *bottom-left position* (abbreviated as *BL position*) relative to the current layout. The BL position of a new item relative to the current layout is defined as the leftmost point among the lowest *bottom-left stable feasible positions*, where a bottom-left stable feasible position is a point such that the new item can be placed without overlap and cannot be moved leftward nor downward.

In this paper, we analyze the strength and weakness of the two algorithms from both sides of the running time and the quality of the packing result. Based on this observation, we then propose a new construction heuristic algorithm *partition-based best-fit heuristic* (abbreviated as *PBF*) as a bridge between the best-fit and bottom-left algorithms. The basic idea of the PBF algorithm is that all the items to be packed are partitioned into groups, and then items are packed into the container in a group-by-group manner. The best-fit algorithm is taken as the internal tactics to pack items of each group. We analyze the time complexity of the PBF algorithm and perform a series of experiments on some benchmark instances. The computational results show that the proposed algorithm is especially effective for large-scale instances of the rectilinear block packing problem.

2. Problem Description

We are given a set of n items $R = \{R_1, R_2, \dots, R_n\}$ of rectilinear blocks, where each rectilinear block takes a deterministic shape and size from a set of t types $T = \{T_1, T_2, \dots, T_t\}$. We are also given a rectangular container C with fixed width W and unrestricted height H. The task is to pack all the items orthogonally without overlap into the container. We assume that the bottom left corner of the container is located at the origin $O = (0, 0)$ with its four sides parallel to x- or y-axis. The objective is to minimize the height H of the container which is necessary to pack all the given items. Note that the minimization of the height H is equivalent to the maximization of the occupation rate defined by $\sum_{i=1}^{n} A(R_i)/WH$, where $A(R_i)$ denotes the area of a rectilinear shape R_i.

We define the bounding box of an item R_i as the smallest rectangle that encloses R_i, and its width and height are denoted as w_i and h_i. We call the area of the bounding box, $w_i h_i$, the *bounding area* of R_i. The location of an item R_i is described by the coordinate (x_i, y_i) of its reference point, where the reference point is the bottom-left corner of its bounding box. For convenience, each rectilinear block and the container C are regarded as the set

1 Department of Computer Science and Mathematical Informatics, Graduate School of Information Science, Nagoya University, Nagoya, Japan
2 Department of Computational Science and Engineering, Graduate School of Engineering, Nagoya University, Nagoya, Japan
a) yannanhu@nagoya-u.jp
b) hasimoto@nagoya-u.jp
c) imahori@nagoya-u.jp
d) yagiura@nagoya-u.jp

of points (including both interior and boundary points), whose coordinates are determined from the origin $O = (0, 0)$. Then, we describe the rectilinear block R_i placed at $v_i = (x_i, y_i)$ by the Minkowski sum $R_i \oplus v_i = \{p + v_i \mid p \in R_i\}$. For a rectilinear block R_i, let $\mathrm{int}(R_i)$ be the interior of R_i. Then the rectilinear block packing problem is formally described as follows:

$$\begin{aligned}
&\text{minimize} && H \\
&\text{subject to} && 0 \le x_i \le W - w_i, \quad 1 \le i \le n && (1)\\
& && 0 \le y_i \le H - h_i, \quad 1 \le i \le n && (2)\\
& && \mathrm{int}(R_i \oplus v_i) \cap (R_j \oplus v_j) = \emptyset,\ i \ne j. && (3)
\end{aligned}$$

The constraints (1) and (2) tell that all the rectilinear blocks must be packed inside the container. The constraint (3) means that there exists no item overlapping with others.

3. Basic Knowledge

In this section, we explain some important techniques and definitions used in our algorithms, and the bottom-left and the best-fit algorithms. As a crucial technique for packing problem, the idea of no-fit polygon is introduced in Section 3.1. As a basic terminology, the BL position is introduced in Section 3.2. Two construction algorithms for the rectilinear block packing problem are introduced in Section 3.3 and 3.4.

3.1 No-Fit Polygon

No-fit polygon (abbreviated as *NFP*) is a geometric technique to check overlaps of two polygons in two-dimensional space. This concept was introduced by Art [7] in 1960s, who used the term "shape envelope" to describe the positions where two polygons can be placed without intersection. It is defined for an ordered pair of two polygons i and j, where the position of polygon i is fixed and polygon j can be moved. $NFP(i, j)$ denotes the set of positions of polygon j having intersection with polygon i, which is formally defined as follows:

$$NFP(i, j) = \mathrm{int}(i) \oplus (-\mathrm{int}(j)) = \{u - w \mid u \in \mathrm{int}(i), w \in \mathrm{int}(j)\}. \quad (4)$$

When the two polygons are clear from the context, we may simply use NFP instead of $NFP(i, j)$. Assume that $\partial NFP(i, j)$ denotes the boundary of $NFP(i, j)$, and the $\mathrm{cl}(NFP(i, j))$ denotes the closure of $NFP(i, j)$. The no-fit polygon has the following important properties:

- $j \oplus v_j$ overlaps with $i \oplus v_i$ if and only if $v_j \in NFP(i, j) \oplus v_i$.
- $j \oplus v_j$ touches $i \oplus v_i$ if and only if $v_j \in \partial NFP(i, j) \oplus v_i$.
- $i \oplus v_i$ and $j \oplus v_j$ are separated if and only if $v_j \notin \mathrm{cl}(NFP(i, j)) \oplus v_i$.

Hence, the problem of checking whether two polygons overlap or not becomes an easier problem of checking whether a point is in a polygon or not.

When i and j are both convex, $\partial NFP(i, j)$ can be computed by the following simple procedure: We first place the reference point of i at the origin $O = (0, 0)$, and slide j around i having it keep touching with i. Then the trace of the reference point of j is $\partial NFP(i, j)$.

3.1.1 Method of Calculating NFP of Rectilinear Blocks

In this paper, we treat rectilinear blocks. However, we only need NFPs between a rectangle and a rectilinear block. Let i be a rectangle and R_j be a rectilinear.

When i and R_j are both rectangles, where rectangle i (resp., R_j) has width w_i (resp., w_j) and height h_i (resp.,h_j), $NFP(i, R_j)$ can be computed by the following expression:

$$NFP(i, R_j) = \{(x, y) \mid -w_j < x < w_i, -h_j < y < h_i\}. \quad (5)$$

When R_j is a rectilinear block, we first divide R_j into a set of rectangles S. For example, a rectilinear block with m_j concave vertices (i.e., vertices whose angle outside of the block is 90°) can be cut into at most $m_j + 1$ rectangular pieces by horizontal lines that go through its concave vertices. For each item S_k in S, we can easily calculate its NFP with respect to i by using (5). The $NFP(i, R_j)$ is the union of these $NFP(i, S_k)$ for all S_k in the set S. The $NFP(i, R_j)$ can be formally calculated as follows:

$$NFP(i, R_j) = \bigcup_{S_k \in S} (NFP(i, S_k) \oplus v_k), \quad (6)$$

where v_k is the position of S_k relative to the reference point of R_j.

3.2 Bottom-Left Position

Bottom-left stable feasible positions are defined for a given area, a set of rectilinear blocks placed in the area, and one new item to be placed. In this paper, we assume that the shape of the given area is rectangular. A bottom-left stable feasible position is a point in the area where the new item can be placed without overlap with already placed rectilinear blocks and the new item cannot be moved leftward nor downward. "Bottom-left stable" means that the new item cannot move to the bottom or to the left, and "feasible" means that the new item will not overlap with other blocks when it is placed.

Note that there are many bottom-left stable feasible positions in general. The BL position is the leftmost point among the lowest bottom-left stable feasible positions.

3.2.1 Method of Calculating BL Position

In this section, we explain how to calculate the BL position of a rectilinear block by using the NFPs. For simplicity, in this section, we assume that all the items in the container are rectangles. This assumption does not lose generality because any rectilinear block can be represented as the union of a set of rectangles.

First we calculate NFPs of the new rectilinear block relative to all the rectangles which are in the container. We then place every NFP at the position where the corresponding rectangle is placed. We define the *CrossPoint* as follows: if one NFP's right edge crosses another's top edge, we call the crossing point as a CrossPoint. Observe that, the bottom-left stable feasible position will only appear at the non-overlapping CrossPoints. By checking such CrossPoints, we can get the bottom-left stable feasible positions. The BL position of the rectilinear block is the leftmost point among the lowest bottom-left stable feasible positions. Instead of checking such CrossPoints one by one, we use the sweep line technique to compute the number of NFPs that cover every CrossPoint. As a result, we can compute the BL position of a rectilinear block R_j in $O(m_j M \log M)$ time, where m_i is the number of rectangles that represent a rectilinear block R_i and $M = \sum_{i=1}^{n} m_i$.

3.3 Bottom-Left Algorithm for Rectilinear Block Packing

In this section, we explain the bottom-left algorithm for the rectilinear block packing problem.

The bottom-left algorithm can be generally explained as follows: We are given a set of n rectilinear blocks R and an order of items (e.g., decreasing order of area). The algorithm packs the items one by one according to the given order, where each item is placed at its BL position of the current layout (i.e., the layout at the time just before it is placed).

3.4 Best-Fit Algorithm for Rectilinear Block Packing

In this section, we explain the best-fit algorithm for the rectilinear block packing problem.

The basic idea of the best-fit algorithm comes from the best-fit algorithm for the rectangle packing problem, which was proposed by Burke et al. [5]. The best-fit algorithm can be generally explained as follows: We are given a set of n rectilinear blocks R and priority among them (e.g., an item with larger area has higher priority). The algorithm packs items one by one, and in each iteration, it dynamically chooses a rectilinear block to pack among the remaining items by the following rule: Calculate the BL positions of all the remaining items. Then a rectilinear block, whose BL position takes the smallest x-coordinate among those with the lowest y-coordinate, is packed in this iteration. If there exists more than one such item, the one with the highest priority among them is chosen.

3.5 Time Complexity

In this section, we explain the time complexity of the bottom-left and best-fit algorithm.

In this paper, we assume that each rectilinear block is represented as a set of rectangles with certain constraints on their relative positions. The number of rectangles that represent a rectilinear block R_i is denoted by m_i, and M denotes the sum of m_i over all the n rectilinear blocks. The sum of m_i of the rectilinear blocks from t distinct types is denoted by m.

The time complexity of both of the bottom-left algorithm and the best-fit algorithm is $O(Mm \log M)$.

4. New Construction Heuristic Algorithm

In this section, we first analyze the performance of the bottom-left algorithm and the best-fit algorithm. Then, we explain our new construction heuristic algorithm PBF, which takes the advantages of these two algorithms. Finally, we analyze the time complexity of our new algorithm.

4.1 Analysis of Bottom-Left Algorithm and Best-Fit Algorithm

According to the experimental results [6] obtained by the bottom-left algorithm and the best-fit algorithm, we observed that the best-fit algorithm performed better with respect to the occupation rate for many of the instances we tested. However, there are also a non-negligible number of instances for which the opposite holds. Observing and analyzing the packing layouts obtained by the two algorithms, we summarize the reason why the best-fit algorithm performs better for many of the instances we tested and for what kind of instances, the bottom-left algorithm performs better.

The reason why the best-fit algorithm performs better for many instances is that whenever the best-fit algorithm packs an item into the container, it tries all the remaining items relative to the current layout, and chooses the one that can be placed at the lowest position. As a result, an item that fits well with the surrounding layout tends to be chosen, which means that redundant space around the new item is usually small. On the contrary, the bottom-left algorithm may not choose a proper item that fits well with the current layout, because the next item to place is always fixed a priori (by the given order of items). Fig. 1 shows an example when the best-fit algorithm performs better. The left layout of Fig. 1 is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height obtained by the bottom-left algorithm is 40 and that obtained by the best-fit algorithm is 37.

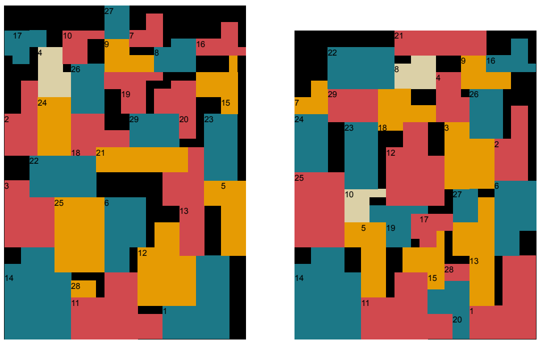

Fig. 1 An example when the best-fit algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

However, there are instances for which the bottom-left algorithm performs better. Recall that the *bounding area* of an item is the area of its bounding box, where the bounding box of an item is the smallest rectangle that encloses the rectilinear block. For example, when there are items whose areas are small but the bounding areas are extremely large, and there are also items whose bounding areas are small, the bottom-left algorithm often performs better.

Note that if an item has a small area but has a large bounding area, it has large blank space inside (i.e., if it is put into its bounding box, large blank space remains in the bounding box in which small items can fit). It is known that the bottom-left algorithm tends to perform well when the given order is the order of decreasing area or decreasing bounding area. Assume that we are given the decreasing order of bounding area. At the beginning of the packing process, the bottom-left algorithm packs relatively bigger items (with greater bounding area) into the container, and some of them have large blank space inside. At this moment, large spaces are often made between such large items. Later, when relatively smaller ones come, they tend to be packed into the blank space between the packed ones. This means that the placement of small ones is not very important, and the height H of the container is mainly decided by the layout of bigger ones.

Conversely, because the BL positions of relatively smaller items are often lower than those of bigger ones, the best-fit algorithm tends to pack them first, and leave relatively bigger ones behind. In the end of the processing of the best-fit algorithm, remaining bigger items are placed on top of smaller ones, and the blank space between these bigger ones significantly reduces the final occupation rate. Fig. 2 shows an example of this case. The left layout of Fig. 2 is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height of the left one is 3960 and that of the right one is 4283.

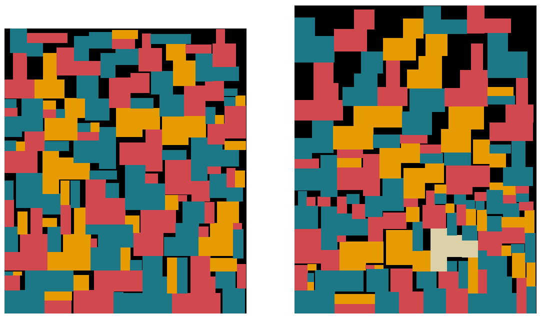

Fig. 2 An example when the bottom-left algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

For the same instance used in Fig. 2, Fig. 3 shows the layouts when the first half of the items are packed into the container. The left layout is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height of the bottom-left algorithm is 3960 and that of the best-fit algorithm is 1880. At this moment, the height of the container obtained by the bottom-left algorithm is already the same as that of its final layout. This means that the remaining half of items have no effect on the final height of the container. On the contrary, many small items have already been packed by the best-fit algorithm, and we can observe by comparing the layouts on the right of Fig. 2 and 3 that most of the remaining items are large.

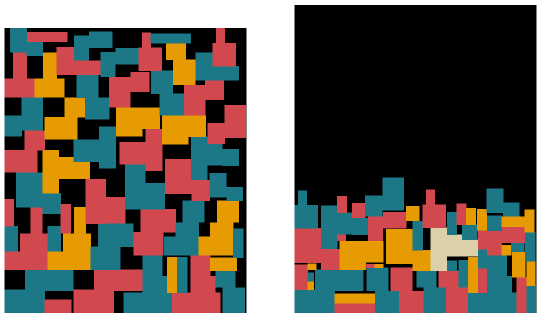

Fig. 3 Layouts when the algorithms pack half of the items (left: the bottom-left algorithm, right: the best-fit algorithm)

4.2 Partition-Based Best-Fit Algorithm

Considering the observation in Section 4.1, the idea of simply choosing either the best-fit algorithm or the bottom-left algorithm according to the property of instances comes naturally. Another simple idea would be just to run both algorithms and then choose the better layout. However, considering the fact that occupation rate of the best-fit algorithm is better in many cases, we propose a new construction heuristic, which uses the best-fit algorithm as its core part, but alleviates the drawback of the best-fit algorithm. The main idea is to divide the items to be packed into groups and then pack the items into the container in a group-by-group manner. We adopt the best-fit algorithm to pack the items of each group. Intuitively, we would like to divide the items into groups so that the smaller the item is, the later the group containing it is processed. There will be many rules that realize this, and the rule we tested is explained in Section 5.

The PBF algorithm is generally explained as follows: We are given a set of n rectilinear items R, which is divided into K groups $B = \{B_1, B_2, \ldots, B_K\}$, and we are also given the priority among the items. The PBF algorithm repeats the following steps:

Step 0. Set $k := 1$.

Step 1. For the current layout, call the best-fit algorithm to pack all the items in group B_k into the container.

Step 2. If $k = K$, output the current layout and stop; otherwise, set $k := k + 1$, and then return to Step 1.

Note that, if $K = 1$, the PBF algorithm performs the same as the best-fit algorithm. If $K = n$, the processing of PBF algorithm is the same as the bottom-left algorithm. If appropriately implemented, the running time of the PBF algorithm is independent of K and is $O(Mm \log M)$. The details of how to realize this time complexity is omitted due to space limitations.

5. Computational Results

The PBF algorithm proposed in this paper was implemented in the C programming language and run on a Mac PC with 2.3 GHz Intel Core i5 processor and 4 GB memory. Performance of the PBF algorithm has been tested on a series of instances, which are generated from nine benchmark instances. For more details of these instances, readers could refer to [8]. The width W of the container is decided by $W = \left\lceil \sqrt{\sum_{i=1}^{n} A(R_i)} \right\rceil$.

For two rectilinear blocks R_i and R_j, $R_i \preceq R_j$ signifies that R_i and R_j can be packed into the bounding box of R_j without overlap. For these instances, we divide all the given items into two groups *Large* and *Small* so that for every item R_i in Small, there exists at least one item R_j in R that satisfies $R_i \preceq R_j$ and $A(R_i)$ is strictly smaller than that of any item in Large. This rule can be formally described as follows:

$$\begin{aligned}
&\text{Tem\_S} = \{R_i \in R \mid \exists R_j \in R, R_i \preceq R_j\}\\
&\text{Tem\_L} = R \setminus \text{Tem\_S}\\
&\text{Small} = \{R_i \in \text{Tem\_S} \mid \forall R_j \in \text{Tem\_L}, A(R_i) < A(R_j)\}\\
&\text{Large} = R \setminus \text{Small}.
\end{aligned}$$

As to the order of the items for the bottom-left algorithm and the priority among the items for the best-fit algorithm, we tested the decreasing order of width, area, and bounding area. The computational results of the decreasing order of area is slightly better than the results obtained by other orders. Hence, we report those results of the decreasing order of area.

The proposed algorithm is tested on a series of instances, which are generated from nine benchmark instances. For some instances, the results obtained by the PBF algorithm and the best-fit algorithm are not much different, and hence, we omit their results and show those of four types of benchmark instances for which interesting outcomes are observed. The computational results obtained by the bottom-left algorithm, the best-fit algorithm, and the

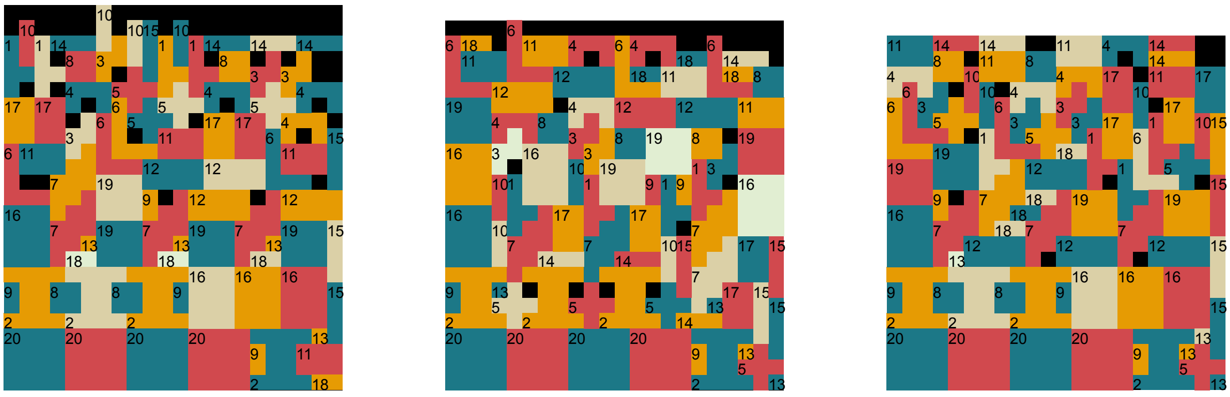
Fig. 4 Layouts obtained for T144_004 by the three algorithms (left: the bottom-left algorithm, middle: the best-fit algorithm, right: the PBF algorithm)

Table 1 Computational results of T144

instance	W	n	BL(%)	BF(%)	PBF(%)
T144_001	11	20	85.31	*92.42	85.31
T144_002	15	40	85.61	*90.37	*90.37
T144_004	22	80	88.73	92.42	*96.44
T144_008	31	160	92.60	*95.41	*95.41
T144_016	44	320	92.42	94.39	*96.44
T144_032	62	640	95.41	96.87	*98.39
T144_064	88	1280	96.44	*97.50	*97.50
T144_128	124	2560	96.87	97.62	*98.39
T144_256	176	5120	98.04	98.59	*99.14
T144_512	249	10240	98.38	98.76	*99.15

Table 2 Computational results of T40

instance	W	n	BL(%)	BF(%)	PBF(%)
T40_001	301	32	*88.72	81.53	86.19
T40_002	426	64	*90.70	87.00	*90.70
T40_004	602	128	*92.82	91.41	*92.82
T40_008	852	256	92.67	93.69	*95.80
T40_016	1205	512	94.19	92.74	*94.93
T40_032	1704	1024	95.26	93.18	*95.80
T40_064	2410	2048	96.07	92.74	*97.23
T40_128	3409	4096	*96.58	93.15	96.31
T40_256	4821	8192	*97.01	93.44	*97.01
T40_512	6818	16384	*96.86	93.53	96.72

Table 3 Computational results of ami49L21

instance	W	n	BL(%)	BF(%)	PBF(%)
ami49L21_001	5936	28	*85.00	84.66	84.49
ami49L21_002	8396	56	83.41	86.42	*88.20
ami49L21_004	11873	112	86.91	86.55	*88.17
ami49L21_008	16792	224	87.68	86.98	*87.94
ami49L21_016	23747	448	88.40	87.17	*89.37
ami49L21_032	33584	896	*89.38	87.78	88.85
ami49L21_064	47495	1792	*89.49	87.82	89.30
ami49L21_128	67168	3584	89.93	88.28	*89.95
ami49L21_256	94991	7168	*90.19	88.84	*90.19
ami49L21_512	134337	14336	*90.59	88.66	90.52

PBF algorithm are shown in Table 1 to 4.

In the tables, the columns of *BL*, *BF* and *PBF* are the occupation rate in % obtained by the bottom-left algorithm, the best-fit algorithm and the PBF algorithm. For each instance, the best results among the three are marked by '$*$'. The computational results show that the PBF algorithm performs best for most of these instances. Even for the instances where the PBF algorithm did not obtain the best results, the difference in the occupation rate is

Table 4 Computational results of ami49LT21

instance	W	n	BL(%)	BF(%)	PBF(%)
ami49LT21_001	5953	27	82.58	*86.80	84.89
ami49LT21_002	8419	54	84.71	87.42	*87.55
ami49LT21_004	11907	108	87.14	86.09	*87.77
ami49LT21_008	16839	216	*88.57	85.31	87.61
ami49LT21_016	23814	432	87.96	86.61	*88.64
ami49LT21_032	33678	864	88.44	87.54	*88.74
ami49LT21_064	47628	1728	89.27	87.41	*89.39
ami49LT21_128	67357	3456	*89.96	87.70	89.81
ami49LT21_256	95257	6912	89.65	87.73	*90.06
ami49LT21_512	134714	13824	90.00	88.38	*90.32

less than 3% between the PBF algorithm and the best one among the other algorithms except for T144_001.

The running times of the bottom-left, the best-fit and the PBF algorithm are 1.89, 2.04 and 1.97 seconds, respectively, for the instance T144_512 with 10,240 items. Fig. 4 shows three example layouts obtained by the bottom-left algorithm, the best-fit algorithm and the PBF algorithm from left to right. These are the layouts obtained for the instance named T144_004. The height obtained by the bottom-left algorithm is 25, that obtained by the best-fit algorithm is 24, and that obtained by the PBF algorithm is 23. The occupation rates of these layouts are reported in Table 1.

Chen et al. [8] proposed an algorithm for a slightly different problem, where the width W of the container is also a variable and the rotation and reflection of items are allowed. Their objective is to maximize the occupation rate. Their algorithm is designed to find a solution of extremely high occupation rate spending long computation time for relatively small instances. For the instance named ami49LT21_001 with 27 items, for example, the occupation rate obtained by their algorithm is 95.09% and W is 6370 with the running time of 2842.75 seconds on an IBM portable PC with 2.0 GHz processor and 512 MB memory. The worst-case time complexity of their algorithm is $O(n^8)$, though they reported that the average time complexity would be much smaller. Indeed, there exists an instance with 29 items for which their algorithm terminated in 2.72 seconds. For the same instance of ami49LT21_001, we tested the PBF algorithm with the value of W every integer from 4000 to 7000. The total running time (of about 3000 calls to the PBF algorithm) was 6.23 seconds and the best occupation rate was 90.14% when $W = 4186$. Note that the rotation and reflection were not considered in our algorithm. The occupation rate was significantly improved from 84.89%, the result reported in Table 4, just by considering various values for W. Although their algorithm obtained higher occupation rate, the running time will not be practical for much larger instances such as those reported in this section. On the contrary, our algorithm is especially efficient for large-scale instances of the rectilinear block packing problem.

6. Conclusions

In this paper, we proposed a new construction heuristic algorithm PBF for the rectilinear block packing problem. We analyzed the time complexity of our algorithm and showed that the PBF algorithm runs in $O(Mm \log M)$ time.

We also performed a series of experiments based on some benchmark instances. The occupation rate of the packing layouts obtained by our algorithm was more than 90% on average for these instances. Even for instances with more than 10,000 rectilinear blocks, the proposed PBF algorithm runs in less than 10 seconds.

References

[1] B.S. Baker, E.G. Coffman Jr. and R.L. Rivest, "Orthogonal packings in two dimensions," SIAM Journal on Computing, vol. 9, pp. 846–855, 1980.
[2] K.A. Dowsland, "Some experiments with simulated annealing techniques for packing problems," European Journal of Operational Research, vol. 68, pp. 389–399, 1993.
[3] J.F. Gonçalves, "A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem," European Journal of Operational Research, vol. 183, pp. 1212–1229, 2007.
[4] Y.L. Wu, W. Huang, S. Lau, C.K. Wong and G.H. Young, "An effective quasi-human based heuristic for solving the rectangle packing problem," European Journal of Operational Research, vol. 141, pp. 341-358, 2002.
[5] E.K. Burke, G. Kendall and G. Whitwell, "A new placement heuristic for the orthogonal stock-cutting problem," Operations Research, vol. 52, pp. 655–671, 2004.
[6] Y. Hu, H. Hashimoto, S. Imahori and M. Yagiura, "Heuristics for the rectilinear block packing problem," Proceedings of the 2012 Spring National Conference of Operations Research Society of Japan, March 27–28, 2012, Yokosuka, Japan, pp. 64–65.
[7] R.C. Art, "An approach to the two dimensional irregular cutting stock problem," IBM Cambridge Science Center, 36.Y08, 1966.
[8] D. Chen, J. Liu, Y. Fu and M. Shang, "An efficient heuristic algorithm for arbitrary shaped rectilinear block packing problem," Computers & Operations Research, vol. 37, pp. 1068–1074, 2010.