Journal of the Operations Research Society of Japan
Vol. 59, No. 1, January 2016, pp. 110–129
© The Operations Research Society of Japan

A PARTITION-BASED HEURISTIC ALGORITHM FOR THE RECTILINEAR BLOCK PACKING PROBLEM

Yannan Hu
Nagoya University

Hideki Hashimoto
Tokyo University of Marine Science and Technology

Shinji Imahori
Chuo University

Takeaki Uno
National Institute of Informatics

Mutsunori Yagiura
Nagoya University

(Received April 9, 2015; Revised August 6, 2015)

Abstract This paper focuses on a two-dimensional strip packing problem where a set of arbitrarily shaped rectilinear blocks need be packed into a larger rectangular container without overlap. A rectilinear block is a polygonal block whose interior angles are either 90°or 270°. This problem involves many industrial applications, such as VLSI design, timber/glass cutting, and newspaper layout. We generalized a *bottom-left* and a *best-fit* algorithms to the rectilinear block packing problem in our previous paper. Based on the analysis of the strength and weakness of these algorithms, we propose a new construction heuristic algorithm called the *partition-based best-fit* algorithm (*PBF* algorithm), which takes advantages of both the bottom-left and the best-fit algorithms. The basic idea of the PBF algorithm is that it partitions all the items into groups and then packs the items in a group-by-group manner. The best-fit algorithm is taken as the internal tactics for each group. The proposed algorithm is tested on a series of instances, which are generated from benchmark instances. The computational results show that the proposed algorithm significantly improves the performance of the existing construction heuristic algorithms and is especially effective for the instances having large differences in the sizes of given shapes.

Keywords: Combinatorial optimization, strip packing, rectilinear block, heuristic, group-manner

1. Introduction

The two-dimensional strip packing problems are complex combinatorial optimization problems that arise in many industries related to materials such as metal, textile, paper, etc. The rectilinear block packing problem is a special case of the problem of packing general polygons, called the irregular packing problem or nesting problem [7]. A rectilinear block is a polygonal block whose interior angles are either 90°or 270°. This paper focuses on the rectilinear block packing problem where a set of arbitrary shaped rectilinear blocks need be packed into a larger rectangular container without overlap. The objective is to minimize or maximize a given objective function. 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 [2].

Several heuristic methods have been proposed for the rectilinear block packing problem based on different data structures to represent the relationships among the blocks, e.g., BSG (bounded sliceline grid) [8, 13], sequence-pairs [5, 9, 10, 17], O-tree [14], B*-tree [16], TCG (transitive closure graph) [11], CBL (corner block list) [12], etc. A special case of the rectilinear block packing problem is the rectangle packing problem. Many efficient algorithms have been proposed to solve the rectangle packing problem, including simulated annealing, hybrid algorithm, and quasi-human heuristic algorithm. The *bottom-left algorithm* [2] and

the *best-fit algorithm* [3] are known as the most remarkable work among existing construction heuristics. Compared with the rectangle packing problem, the rectilinear block packing problem is more complicated, and it is difficult to design efficient data structures to represent the relative relationships among the rectilinear blocks. Inspired by the representative construction heuristics for rectangle packing, we generalized the bottom-left and the best-fit algorithm for the rectangle packing problem to the case of the rectilinear block packing problem, and also designed their efficient implementations in [6].

The main strategy of the bottom-left and the best-fit algorithms is the *bottom-left strategy*, which is proposed in [2]. In this strategy, whenever a new item is packed into the container, it is 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 first analyze the strength and weakness of the bottom-left and the best-fit algorithms from the viewpoints of the running time and the quality of the packing results. We then summarize the reasons why the best-fit algorithm outperforms the bottom-left algorithm for many instances and situations when the bottom-left algorithm performs better for some kinds of instances. Based on these observations, we propose a new construction heuristic algorithm called the *partition-based best-fit heuristic* (abbreviated as *PBF*) as a bridge between the bottom-left and the best-fit 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 also show that the PBF algorithm runs in the same time complexity as the efficient implementations in [6] of the bottom-left and the best-fit algorithms (these implementations in [6] of the two algorithms have the same time complexity). We also give some effective rules to partition items into groups.

We perform a series of experiments on instances that were generated from nine benchmark instances. The computational results show that the proposed PBF algorithm significantly improves the performance of the bottom-left and the best-fit algorithms and is especially effective for instances having large differences in the sizes of given shapes.

2. Problem Description

We are given a set of n items $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of rectilinear blocks, where each rectilinear block takes a deterministic shape and size from a set of t shapes $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$. 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 that 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 block R_i. This type of problem is often called the strip packing problem (e.g., the rectangular strip packing problem for the rectangular case and the irregular strip packing problem for the case of general polygons), and according to the improved typology of Wäscher et al. [15], strip packing problems are categorized into the two dimensional open dimension problem (2D ODP) with a single variable dimension. Figure 1 shows an example of the rectilinear block packing problem. The layout on the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

right is an example packing layout after packing all the rectilinear blocks into the container given on the left of Figure 1. The number of rectilinear blocks n is 7, and that of shapes t is 6. The task is to pack these seven items into the rectangular container so as to minimize the height of the container.

Figure 1: An instance of the rectilinear block packing problem and a solution

We define the bounding box of 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 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 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:

$$
\begin{array}{llrr}
\text{minimize} & H & & \\
\text{subject to} & 0 \leq x_i \leq W - w_i, & 1 \leq i \leq n & (2.1) \\
 & 0 \leq y_i \leq H - h_i, & 1 \leq i \leq n & (2.2) \\
 & \mathrm{int}(R_i \oplus v_i) \cap (R_j \oplus v_j) = \emptyset, & i \neq j. & (2.3)
\end{array}
$$

The constraints (2.1) and (2.2) tell that all the rectilinear blocks must be packed inside the container. The constraint (2.3) means that there exists no item overlapping with others.

Two cases of this problem are often considered in the literature: (1) all the items are not allowed to be rotated, and (2) all the items can be rotated 90°, 180° or 270°. In this paper, we concentrate on the case without rotations.

3. Basic Knowledge

In this section, we explain the definitions used in our algorithm and two construction algorithms for the rectilinear block packing problem. As a basic terminology, the definition of BL position in general is introduced in Section 3.1. The bottom-left and the best-fit algorithms for the rectilinear block packing problem are introduced in Section 3.2 and 3.3. The bottom-left and the best-fit algorithms are known as the most remarkable work among the existing construction algorithms for the rectangle packing problem. We generalized these two construction algorithms for the rectilinear block packing problem and designed their efficient implementations in [6]. We first explain the bottom-left algorithm, which is one of the simplest forms among the algorithms based on the bottom-left strategy. Then we explain the best-fit algorithm, which is slightly more complicated than the bottom-left

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

algorithm but it is known to be more effective. We also give the explanation of the time complexities of these two algorithms in Section 3.4.

3.1. Bottom-left stable feasible positions

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. 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 *bottom-left position* (*BL position*) is defined as the leftmost location among the lowest bottom-left stable feasible positions. The bottom-left stable feasible positions and the BL position are illustrated through the example in Figure 2.

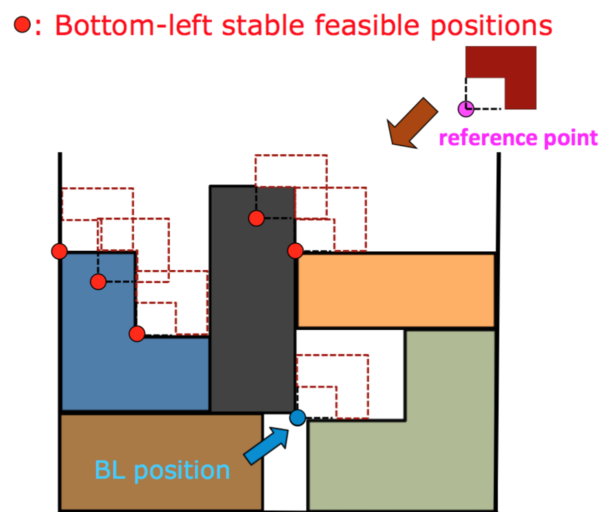

Figure 2: Bottom-left stable feasible positions and the BL position

3.2. Bottom-left algorithm for the rectilinear block packing problem

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

The bottom-left algorithm can be generally explained as follows: Given a set of n rectilinear blocks $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and an order of items (e.g., decreasing order of area), the algorithm packs all of the items one by one according to the given order, where each item is placed at its BL position relative to the current layout (i.e., the layout at the time just before it is placed).

3.3. Best-fit algorithm for the rectilinear block packing problem

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

The best-fit algorithm for the rectilinear block packing problem is explained as follows: Given a set of n rectilinear blocks $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and a priority among them (e.g., an item with a wider bounding box has higher priority), the algorithm packs all of the items one by one into the container, where each item is placed at its BL position relative to the current layout. At the beginning of the packing process, no item is placed in the container. Whenever an item is to be packed into the container, the algorithm calculates the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

BL positions of all of the remaining items relative to the current layout. In this iteration, the rectilinear block whose BL position takes the smallest x-coordinate among those with the lowest y-coordinate is packed. In case of ties, we choose the block with the highest priority among those whose BL positions take the same coordinates.

3.4. Time complexities

In this section, we explain the time complexities of the bottom-left and the best-fit algorithms for the rectilinear block packing problem that are implemented with the methods proposed in [6].

We assume that each rectilinear block is represented as the union of a set of rectangles whose relative positions are fixed, and let m_i be the number of rectangles that represent a rectilinear block R_i. We also assume that each such rectangle has a positive area, i.e., special cases of rectangles such as line segments and points are not considered. Note that a rectilinear block with r_i concave vertices (i.e., vertices whose angles outside of the block are $90°$) can be cut into at most $r_i + 1$ rectangular pieces by horizontal lines that go through its concave vertices. Hence, a rectilinear block with r_i concave vertices can be represented by at most $r_i + 1$ rectangles, i.e., $m_i \leq r_i + 1$. It is also noted that there is no restriction on the way the relative positions are fixed, e.g., an item R_i can be a set of two separate rectangles as long as their relative positions are fixed. Hence, these algorithms can also deal with the packing problem in which each item can be a set of (non-overlapping) rectilinear blocks whose relative positions are fixed.

As explained before, the number of rectangles that represent a rectilinear block R_i is denoted by m_i, and let $M = \sum_{i=1}^{n} m_i$. Recall that each rectilinear block takes a deterministic shape from the set of t shapes $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$, and when $t < n$, some items have an identical shape. We define m_j^{T} as the number of rectangles that represent shape T_j, and m is the sum of m_j^{T} for all of the shapes in $\mathcal{T}$. Both the bottom-left and best-fit algorithms run in $O(mM \log M)$ time with the efficient implementations proposed in [6].

4. New Construction Heuristic Algorithm

In this section, we propose a new partition-based heuristic algorithm for the rectilinear block packing problem. We first analyze the performance of the bottom-left algorithm and the best-fit algorithm in Section 4.1. According to these observations, in Section 4.2, we propose a new construction heuristic algorithm, the PBF algorithm, which takes advantages of both the bottom-left and the best-fit algorithms. Then, in Section 4.3, we analyze the time complexity of our new algorithm. Finally, in Section 4.4, we give some effective partition rules that are used to partition items into groups for the PBF algorithm.

4.1. Analysis of the performance of the bottom-left and best-fit algorithms

In this section, we analyze the performance of the bottom-left and the best-fit algorithms according to the experimental results addressed in [6].

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 these two algorithms, we summarize the reason why the best-fit algorithm performs better for many of the instances we tested and the situations when 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.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

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). Figure 3 shows an example when the best-fit algorithm performs better. The left layout of Figure 3 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.

Figure 3: An example when the best-fit algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

However, for some cases, e.g., when there are items whose areas are small but the bounding areas are large, and there are also items whose bounding areas are small, the bottom-left algorithm often performs better. Note that if the area of an item is small but its bounding area is large, 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 sizes of items, where various measures for sizes can be considered such as the areas of items, their bounding areas, widths or heights. Let us consider the case where the algorithm packs items in 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 does not have much influence on the final value of height H of the container, and H 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 smaller ones first, and leave relatively bigger ones behind. In the end of the processing of the best-fit algorithm, the remaining bigger items have no choice but to place on the top of smaller ones, and the blank space between these bigger items have significantly negative effect on the final occupation rate. Figure 4 shows an example of this case. The left layout of Figure 4 is obtained by the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

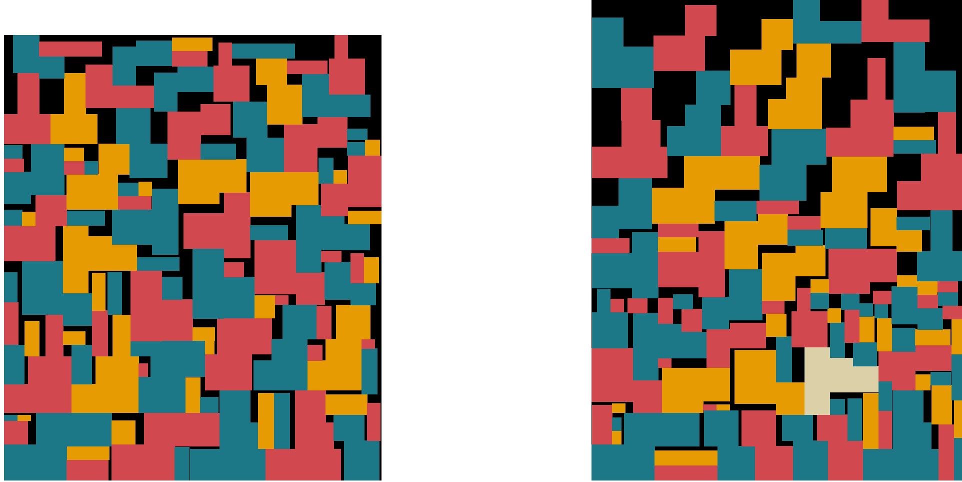

Figure 4: An example when the bottom-left algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

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.

For the same instance used in Figure 4, Figure 5 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 left one is 3960 and that of the right one 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 suggests 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 by comparing the layouts on the right of Figures 4 and 5, we can observe that most of the remaining items are large.

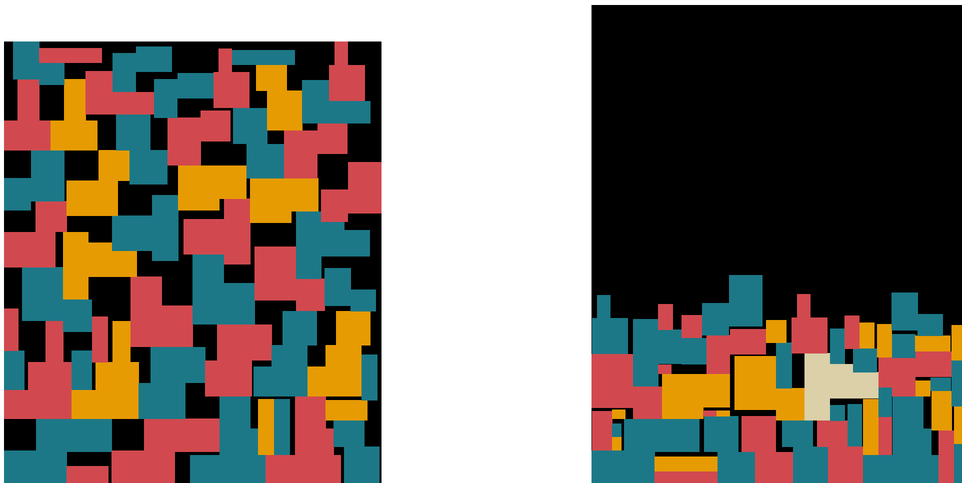

Figure 5: Layouts when the algorithms pack half of the items (left: the bottom-left algorithm, right: the best-fit algorithm)

4.2. Partition-based best-fit algorithm

In this section, we propose a new construction heuristic algorithm, the partition-based best-fit (PBF) algorithm, for the rectilinear block packing problem.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

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, note the fact that the best-fit algorithm performs excellent in many cases when the sizes of items are similar. Hence, we propose a new construction heuristic algorithm, 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 given rectilinear blocks into groups and then pack the items into the container in a group-by-group manner. We utilize the best-fit algorithm to pack the items in each group. Intuitively, we would like to divide the items into groups so that the sizes of items in the same group are similar. There will be many rules to achieve this, regarding how to measure the sizes, how to divide items into groups and so forth. The rules we consider for measuring sizes and for dividing items into groups will be explained in Section 4.4.

Because there are many possible rules to divide the items into groups and to give priority among the items, we explain the framework of the PBF algorithm assuming that we are given a partition and priority among items. Then the PBF algorithm is generally explained as follows: We are given a set of n rectilinear items $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, which is divided into K groups $\mathcal{B} = \{B_1, B_2, \ldots, B_K\}$, and the priority among the items. The PBF algorithm packs all of the groups one by one according to the given order, where each group is packed by the best-fit algorithm relative to the current layout (i.e., the layout at the time just before it is placed). The PBF algorithm is formally described as Algorithm 1.

Algorithm 1 PBF algorithm

1: Set $k := 1$.
2: For the current layout,
call the best-fit algorithm to pack all the items in group B_k into the container.
3: If $k = K$, output the current layout and stop;
otherwise, set $k := k + 1$, and then return to Step 2.

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. In this sense, the PBF algorithm generalizes the bottom-left and the best-fit algorithms, bridging the gap between them.

4.3. Time complexity of PBF algorithm

In this section, we analyze the time complexity of the PBF algorithm. As explained in Section 3.4, m_i is the number of rectangles that represent a rectilinear block R_i and m_j^{T} is the number of rectangles that represent shape T_j. Recall also that $M = \sum_{i=1}^{n} m_i$ and $m = \sum_{j=1}^{t} m_j^{\mathrm{T}}$.

We can implement the PBF algorithm so that is runs in the same time complexity as the best-fit algorithm, which equals $O(mM \log M)$ time, by slightly modifying the efficient implementations proposed in [6].

Below we briefly explain the basic idea of the efficient implementation in [6] of the best-fit algorithm. We utilized the technique of no-fit polygon (*NFP*) [1] to check overlaps among items and to compute the BL positions of items. The NFP of an item R_j relative to R_i has the following property: the reference point of R_j is contained in the NFP if and only if R_j overlaps with R_i. When the algorithm computes the BL position of an item R_j, it uses the NFPs of R_j relative to the items in the container, and such NFPs are placed at the positions

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

where the corresponding items are placed. We call such a layout of NFPs an *NFP layout* for R_j. One of the advantages of such a layout is that the problem of finding the BL position of R_j reduces to the problem of finding the leftmost position among the lowest positions that are not contained in any of the NFPs in the NFP layout. Note that if the shape of two items R_j and $R_{j'}$ are the same, their NFP layouts are the same. Hence, we only need to keep $t = |\mathcal{T}|$ NFP layouts, each for a distinct shape in $\mathcal{T}$, to compute BL positions for the remaining items. A common feature of construction heuristics is that once an item is packed into the container, its position is fixed and will not change. This indicates that it is not necessary to compute NFP layouts from scratch in each iteration of the construction heuristics. The basic idea is to dynamically keep the NFP layout with respect to the current packing layout for each shape in $\mathcal{T}$ during the packing process. Whenever an item is to be placed into the container, the algorithm computes the BL position of every shape T_j by using the NFP layout for T_j. It then chooses an item R_i to place in this iteration (i.e., the item whose BL position is the leftmost among the lowest) and place it at its BL position. The algorithm then updates the NFP layout of every shape T_j in $\mathcal{T}$, adding to the NFP layout the NFP of T_j relative to R_i. It is shown in [6] that throughout the entire computation of the best-fit algorithm, the total running time for computing BL positions of shape T_j is $O(m_j^{\mathrm{T}} M \log M)$, including the time to update the NFP layout of T_j. The total computation time of this part for all shapes is therefore $\sum_{j=1}^{t} O(m_j^{\mathrm{T}} M \log M) = O(mM \log M)$. This dominates the running time of the other parts of the algorithm. As a consequence, the best-fit algorithm runs in $O(mM \log M)$ time.

The point is that, with this implementation, the BL positions of all the remaining items are available in every iteration. We can therefore implement the PBF algorithm similarly, just by considering in each iteration, the items belonging to the current group to be packed in this iteration, instead of considering all the remaining items in choosing the next item to place. Such a modification does not increase the execution time from that of the best-fit algorithm. Hence, with an implementation similar to the best-fit algorithm, the PBF algorithm runs in $O(mM \log M)$ time.

It is easy to apply this result to the case with rotations. Because there are only four possible angles, 0°, 90°, 180° and 270°, all we have to do is to prepare four NFP layouts for every shape T_j, depending on the angle it is to be placed, to compute for every angle, the BL position of T_j when it is placed with the specified angle. Then the time complexity becomes four times that of the case without rotations, and hence we can conclude that the case with rotations can be handled with the same time complexity as the case without rotations.

4.4. Partition rules

In this section, we explain some rules that are utilized to partition the given items into groups. Our rules are based only on the shapes and sizes of items, and for this reason, any two rectilinear blocks with the same shape and size are always in the same group. Hence the partition of items can be defined by a partition of shapes in $\mathcal{T}$. We consider a framework in which the PBF algorithm (Algorithm 1) is repeatedly called with a series of partitions. Algorithm 1 is first applied to the partition consisting of only one block. Then in each iteration, Algorithm 1 is applied to a partition generated by dividing a group in the partition for the previous iteration into two (i.e., in each iteration, the number of groups in a partition increases by one). We explain the rules of how to partition a group into two in Section 4.4.1, and the rules to choose a group to be divided in Section 4.4.2. We test the PBF algorithm on a series of instances based on these rules and the computational results will be addressed in Section 5.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

4.4.1. Rules to partition a group

Recalling the analysis of the performance of the bottom-left and the best-fit algorithm discussed in Section 4.1, the essential purpose of our rules is to pack at the early stage, the items that are "difficult" for the algorithm to pack or that have "negative" impact on the final occupation rate. We propose two types of rules, the *static* rules and the *adaptive* rules, to measure such "difficulty" or "negative impact" of the items.

The static rules divide a given group in two parts according to the information of properties of the items themselves. We consider various rules to partition one group. Considering the analysis that is explained in Section 4.1, we propose *size-based* rules in which the sizes are measured by the values of the areas, bounding areas, widths or heights of the items. First we sort the items in the given group in decreasing order of size with respect to a criterion (e.g., area) and then divide the group into two at the place between two adjacent items in the sorted list where the gap with respect to the size criterion is the largest. For example, assume that there is a group with five shapes of rectilinear blocks $\mathcal{G} = \{T_1, T_2, T_3, T_4, T_5\}$ whose areas are "$12, 2, 10, 5, 11$," respectively. If we use the size-based rule with area for the criterion, we sort the shapes in $\mathcal{G}$ in the decreasing order of areas and obtain the sequence "T_1, T_5, T_3, T_4, T_2" and the corresponding sequence of areas "$12, 11, 10, 5, 2$." The gap between the first and second items is $12 - 11 = 1$, that of the second and third is $11 - 10 = 1$, and so forth. We divide the group $\mathcal{G}$ between the third and fourth items where the biggest gap of 5 is achieved, obtaining two new groups $\mathcal{G}_1 = \{T_1, T_3, T_5\}$ and $\mathcal{G}_2 = \{T_2, T_4\}$.

We also consider another static rule named *inclusion-based* rule. For two shapes T_i and $T_j \in \tilde{\mathcal{T}}$ where $\tilde{\mathcal{T}}$ is the current group to partition, $T_i \preceq T_j$ signifies that T_i and T_j can be packed into the bounding box of T_j without overlap. We divide the items in the given group into two parts LARGE and SMALL so that for every item T_i in SMALL, there exists at least one item T_j in the given group that satisfies $T_i \preceq T_j$. This rule can be formally described as follows:

$$\begin{aligned} \textsc{Small} &= \{T_i \in \tilde{\mathcal{T}} \mid \exists T_j \in \tilde{\mathcal{T}}, T_i \preceq T_j\}, \\ \textsc{Large} &= \tilde{\mathcal{T}} \setminus \textsc{Small}. \end{aligned}$$

The adaptive rules utilize the information of the BL positions of the rectilinear blocks in the given group relative to a packing layout obtained in the current iteration (i.e., the latest call to Algorithm 1), in order to generate a partition for the next iteration. In computing the BL positions that are utilized for this purpose, we consider two strategies and call the resulting rules *BL-midway* and *BL-final*. In the rule of BL-midway, for each group B_k, we recompute the BL position and store it for every shape T_i in B_k relative to the packing layout at the time immediately after all the rectilinear blocks in group B_k have been packed into the container. (Note that at the time such BL positions are recomputed, all items with shape T_i for every shape T_i in B_k have already been packed, and such recomputed BL positions will not be used to actually place rectilinear blocks, but just to generate a partition for the next iteration. Note also that we do not have to actually recompute BL positions, because they are automatically obtained whenever an item is placed and the NFP layouts are updated, and hence the recomputation of BL positions will not increase the time complexity.) In the rule of BL-final, we recompute the BL position and store it for every shape T_i relative to the packing layout after packing all the rectilinear blocks into the container (i.e., we store the BL position of every shape relative to the final packing layout). Then, we sort the shapes in the group to divide in the decreasing order of the y-coordinates of their recomputed BL positions. We then divide the group into two at the place between two adjacent items in

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

the sorted order where the difference of y-coordinates is largest.

4.4.2. Rules to choose a group to divide

In this section, we give some rules to choose a group to divide next.

Considering the groups that are obtained by the inclusion-based rule, only the group of Small could be divided. Indeed, with this rule, the Large group cannot be divided any more for the following reason. For every pair of shapes T_i and T_j in Large, $T_i \preceq T_j$ cannot hold because otherwise, $T_i \preceq T_j$ and $T_j \in \text{Large} \subseteq \tilde{\mathcal{T}}$ would imply that T_i must have been in Small, which is a contradiction. Hence, we always choose Small for the group to divide next. This partition process terminates when there is less than two shapes in Small.

For the other rules we explained in Section 4.4.1, we propose four rules to choose the group to divide next time. When these rules are used, partition process terminates when the chosen group contains a unique shape.

We first give three simple rules labeled *FirstGrp*, *LastGrp* and *LargeGrp* as follows. Below we assume that groups are sorted according to the order of items with respect to the adopted criterion. For example, assuming that the decreasing order of area is considered, when a group is divided into two, the group containing items with larger area is listed first, and the two new groups are placed at the place of the original group (before the division is applied) in the list of groups.

- In the FirstGrp rule, we always choose the first group that contains more than one shape.
- In the LastGrp rule, we always choose the last group that contains more than one shape.
- In the LargeGrp rule, we choose the group with the largest size (i.e., the group that contains the largest number of shapes). If there is more than one group with the largest size, we break the tie by choosing the first one among them.

Considering the analysis of the performance of the bottom-left and the best-fit algorithm, one of our intentions of partitioning the rectilinear blocks into groups is to make the best-fit algorithm perform well in packing each group into the container. To achieve this, it is preferable that the sizes of items in the same group are similar. Hence, whenever we choose a group to divide, it is natural to choose a group that contains shapes with large differences in their sizes. Recall that in the rules of Section 4.4.1 to partition one group into two (except for the inclusion-based rule), the group is always divided at a place between two adjacent items in the sorted order where the gap with respect to the considered criterion is the largest. The value of this gap will have correlation with the differences among the sizes of items in one group. As a consequence, we consider another rule named *BigGapGrp* to choose the group to divide as follows.

- In the BigGapGrp rule, we choose the group with the biggest gap, where the gap is the one used by an adopted partition rule. If there exist ties, the group with the largest size is chosen.

As explained at the beginning of this section, any two rectilinear blocks with the same shape are in the same group. This indicates that the PBF algorithm terminates after at most $t-1$ iterations, if we partition one group into two in each iteration. Note that, when we begin with one group, the processing of the PBF algorithm is the same as the best-fit algorithm, and after $t-1$ iterations, when items are divided into t groups of distinct shapes, it becomes the same as the bottom-left algorithm.

5. Computational Results

The PBF algorithm proposed in this paper was implemented in the C programming language and run on a Mac PC with a 2.3 GHz Intel Core i5 processor and 4 GB memory. Performance

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

of the PBF algorithm has been tested on a series of instances, which are generated from nine benchmark instances. The information of these benchmark instances is addressed in Table 1. For more details of these instances, readers could refer to [4]. We generate a set of instances named *C-class* by copying every shape in these nine instances. Table 2 shows the information of these classes of instances. For example, the set of instances generated with this rule from the instance "ami49L21" is labeled "C-ami49L21". For analyzing the performance of the algorithms for the case when the sizes of rectilinear blocks are quite different, we also test the bottom-left, the best-fit and the PBF algorithms on the another set of instances named *E-class* that are generated by adding enlarged shapes of the instances in Table 2. Each rectilinear block in an instance of E-class is generated by enlarging the size of one item in the corresponding instance in Table 2 by one, two, four or eight times. The information of the instances in E-class is addressed in Table 3. The width W of the container is decided by $W = \left\lceil \sqrt{\sum_{i=1}^{n} A(R_i)} \right\rceil$. The column of "#inst" reports the number of instances in each class. All of these instances are available at http://www.co.cm.is.nagoya-u.ac.jp/~yagiura/rectilinear/.

Table 1: Information of benchmark instances

Name	t	n	m	M
ami49L21	28	28	49	49
ami49LT21	27	27	49	49
TMCNCGSRC	51	51	76	76
B10	9	9	14	14
B30	29	29	59	59
T19	19	19	42	42
T40	32	32	42	42
T64	15	15	33	33
T144	20	20	31	31

Table 2: Information of C-class instances that are generated by copying benchmark instances

Name	#inst	t	n	m	M	W
C-ami49L21	10	28	28–14336	49	49–25088	5936–134337
C-ami49LT21	10	27	27–13924	49	49–25088	5953–134714
C-TMCNCGSRC	9	51	51–13056	76	76–19456	2376–38018
C-B10	12	9	9–18432	14	14–28672	9–438
C-B30	10	29	29–14848	59	59–30208	29–676
C-T19	11	19	19–19456	42	42–43008	18–601
C-T40	10	32	32–16384	42	42–21504	301–6818
C-T64	11	15	15–15360	33	33–33792	7–249
C-T144	10	20	20–10204	31	31–15872	11–249

As for the order of items for the bottom-left algorithm and the priority among items for the best-fit algorithm, we tested the decreasing order of height, 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. We define the *BestOccup* as the best occupation rate of an instance found when the processing terminates. The processing of the PBF algorithm begins with one group and is repeated until no further partition is possible for the chosen group. *AvgOccup* is the average value of BestOccup over all instances in each class.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 3: Information of E-class instances that are generated by adding enlarged copies of shapes of different sizes

Name	#inst	t	n	m	M	W
E-ami49L21	8	112	112–14366	196	196–25088	54735–619265
E-ami49LT21	8	108	108–13824	196	196–25088	54889–621004
E-TMCNCGSRC	7	204	204–13056	305	305–19456	21906–175255
E-B10	10	36	36–18432	56	56–28672	89–2022
E-B30	8	116	116–14848	236	236–30208	275–3118
E-T19	9	76	76–19456	168	168–43008	173–2771
E-T40	8	128	128–16384	168	168–21504	2778–31430
E-T64	9	60	60–15360	132	132–33792	72–1152
E-T144	8	80	80–10240	124	124–15872	101–1152

The occupation ratios obtained by the bottom-left, best-fit and the PBF algorithms are shown in Table 4 and 5. The average for all instances in each set of instances is reported. The columns of *BL* and *BF* show the avgOccup in percent obtained by the bottom-left and the best-fit algorithms. The column of Incl-based is the avgOccup in percent obtained by the PBF algorithm in which we utilize the Inclusion-based rule and the priority among items is set to the decreasing order of area. For each set of instances, the best results among the seven algorithms are marked by '$*$'.

In Table 4, we address the computational results obtained by the PBF algorithm when we use the rules of FirstGrp, LastGrp, LargeGrp and BigGapGrp for choosing a group to divide next. The columns of FirstGrp, LastGrp, LargeGrp and BigGapGrp are the avgOccup in percent of each class obtained by the PBF algorithm when utilizing the rules of FirstGrp, LastGrp, LargeGrp and BigGapGrp to choose a group to divide. For each instance set (i.e., for each row), the table shows the average of the best result obtained for each instance by multiple calls to the PBF algorithm with different rules in Section 4.4.1 to divide one group (except for the Inclusion-based rule). For example, the value of "90.24%" in the cell of row C-ami49L21 and column FirstGrp is the average of the values of BestOccup for all the instances in class C-ami49L21, where the BestOccup of every instance is the best occupation rate when we try all of the Size-based and the adaptive rules to partition the first group. The details of the computational results for each class are reported in Appendix A.

We also summarize the results obtained when using the different rules to partition one group in two. We report the computational results obtained when we use the rules of Size-based and the adaptive rules to partition one group in Table 5. The columns of *BndArea* (bounding area), *Area*, *Width*, *Height* in *Static* and *BL-midway* and *BL-final* in *Adaptive* are the avgOccup in percent of the BestOccup that are obtained by the PBF algorithm when we fix these rules to partition one group and examine all the rules introduced in Section 4.4.2 for choosing the group to divide next. For example, the value of "89.58%" in the cell of row C-ami49L21 and column BndArea is the average of the values of BestOccup for all the instances in class C-ami49L21, where the BestOccup of every instance is the best occupation rate when we try all of rules to choose one group and divide it according to the values of the corresponding criterion from bounding areas of the items. We use the decreasing order of area as the priority among the items when adaptive rules are utilized. If the Size-based rules are used to partition one group, the priority among the items is decided by the decreasing order of the values of the bounding areas, areas, widths and heights of the items.

The computational results show that the PBF algorithm improves the occupation rate

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 4: Comparison of computational results with a fixed rule to choose the group to divide next

Instance	BL	BF	Incl-based	FirstGrp	LastGrp	LargeGrp	BigGapGrp
C-ami49L21	88.10	87.32	88.51	*90.24	89.97	89.99	90.00
C-ami49LT21	87.83	87.10	88.79	90.26	89.82	90.03	*90.26
C-TMCNCGSRC	86.92	81.71	88.93	*90.18	90.05	90.17	90.00
C-B10	87.66	90.32	90.59	*92.06	*92.06	91.74	91.74
C-B30	84.52	88.14	88.31	*89.42	89.25	89.18	89.19
C-T19	73.79	76.24	76.61	*78.55	78.35	78.35	78.41
C-T40	94.09	91.24	94.33	*96.62	96.30	96.30	96.54
C-T64	86.99	92.92	92.92	*93.50	*93.50	*93.50	*93.50
C-T144	91.11	93.39	96.40	*96.90	96.60	96.60	96.45
E-ami49L21	94.71	89.51	94.86	95.65	*95.70	95.69	95.69
E-ami49LT21	94.72	87.66	94.63	95.46	95.62	*95.70	95.62
E-TMCNCGSRC	92.89	83.55	93.18	93.58	93.94	*94.08	93.88
E-B10	93.93	87.92	94.75	96.10	96.55	*96.71	96.69
E-B30	94.58	89.20	95.27	95.55	95.59	95.63	*95.90
E-T19	86.95	84.19	88.42	88.94	*89.69	89.58	89.57
E-T40	98.01	90.63	98.33	98.80	98.88	*98.94	98.90
E-T64	95.33	92.87	96.42	97.19	97.49	97.48	*97.54
E-T144	97.83	96.71	98.32	98.74	98.73	*98.95	98.73

Table 5: Comparison of computational results with a fixed rule to divide a group

			Static					Adaptive	
Instance	BL	BF	Incl-based	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	88.51	89.58	89.75	89.28	89.50	90.12	*90.23
C-ami49LT21	87.83	87.10	88.79	89.76	89.99	88.92	89.47	90.13	*90.30
C-TMCNCGSRC	86.92	81.71	88.93	89.93	90.08	87.67	88.26	*90.15	90.07
C-B10	87.66	90.32	90.59	90.36	91.28	90.25	91.15	*91.68	*91.68
C-B30	84.52	88.14	88.31	88.30	88.81	88.63	88.42	*89.16	88.98
C-T19	73.79	76.24	76.61	76.63	77.16	77.66	75.67	*77.87	77.49
C-T40	94.09	91.24	94.33	96.16	96.13	94.88	*96.30	95.29	95.21
C-T64	86.99	92.92	92.92	*93.43	92.92	93.24	92.87	92.92	92.92
C-T144	91.11	93.39	*96.40	96.30	96.14	95.21	95.30	95.72	95.01
E-ami49L21	94.71	89.51	94.86	95.34	95.51	95.06	95.32	*95.72	95.70
E-ami49LT21	94.72	87.66	94.63	95.29	95.39	95.00	95.30	*95.65	95.62
E-TMCNCGSRC	92.89	83.55	93.18	*94.09	93.83	92.26	92.95	93.84	93.60
E-B10	93.93	87.92	94.75	95.58	95.48	95.16	*96.61	95.98	95.93
E-B30	94.58	89.20	95.27	95.39	95.44	94.89	94.91	*95.61	95.57
E-T19	86.95	84.19	88.42	87.64	89.22	87.49	88.86	89.23	*89.36
E-T40	98.01	90.63	98.33	98.90	98.74	97.64	*98.92	98.71	98.70
E-T64	95.33	92.87	96.42	96.91	96.53	96.42	*97.41	97.31	96.98
E-T144	97.83	96.71	98.32	98.72	98.64	98.31	*98.82	98.30	97.87

significantly compared with the bottom-left and the best-fit algorithms, and the improvement is more remarkable for the case where the sizes among the rectilinear blocks are much different (i.e., the instances in E-classes).

The results in Table 4 show that the PBF algorithm performs best for the instances whose sizes are not much different (i.e., instances in C-classes) when the algorithm chooses the group to divide with the FirstGrp rule. If there are large differences in the sizes of the rectilinear blocks (i.e., instances in E-classes), the PBF algorithm performs better for most of such instances when the algorithm chooses the group to divide with the LargeGrp rule.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Note that even for the cases where the PBF algorithm did not obtain the best results with the above rules, the difference in the occupation rate is less than 0.77% between the results obtained by the best and the worst rules among the four rules (i.e., FirstGrp, LastGrp, LargeGrp and BigGapGrp). This suggests that the performance of the PBF algorithm is robust against the rules of choosing one group to divide.

The big difference among the results in Table 5 indicates that the rules to partition one group into two are very important for the PBF algorithm. The PBF algorithm performs better for most of the instances when using the adaptive rule BL-midway or the static rule Height. Even for the classes where the occupation rates obtained by these two rules are not the best, the difference between the best of these two rules and the best among all rules is less than 0.68%.

We report the running time spent by the bottom-left, the best-fit and the PBF algorithms in Table 6 to 8. All the running times are shown in seconds. Table 6 shows the running time spent for the instance with the largest size in every class. We also report the running times for instances with various sizes. For this purpose, we choose TMCNCGRSC because its computation times in Table 6 are the longest for both of the categories of the C- and E-classes. Table 7 and 8 address the running times for instances in classes C-TMCNCGSRC and E-TMCNCGSRC, respectively. The columns of *BL-Time* and *BF-Time* show the running times of the bottom-left and the best-fit algorithms. The running time of the PBF algorithm when using the Inclusion-based rule to partition the SMALL group are addressed in column *Incl-time*. For other pairs of rules to choose and partition a group (e.g., the combination of LargeGrp and BL-final), the PBF algorithm is repeated $t-1$ times, and we consider the total computation time for the $t-1$ iterations. The combination of such rules do not have a large influence on the computation time except that the PBF algorithm with the adaptive rule BL-midway and BL-final tend to spend slightly more computation time. For this reason, we do not report the results obtained by all pairs of rules but just report the running time of the PBF algorithm with the combination of LargeGrp and BL-final rules in column *Largest-Final*.

Table 6: Computation time for the largest instances in each class

Instance	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
C-ami49L21_512	28	14336	49	25088	5.12	6.52	18.26	147.64
C-ami49LT21_512	27	13924	49	25088	5.51	7.00	22.17	166.96
C-TMCNCGSRC_256	51	13056	76	19456	6.23	8.91	45.64	380.30
C-B10_2048	9	18432	14	28672	1.52	1.82	3.64	12.56
C-B30_512	29	14848	59	30208	8.25	9.13	31.45	247.60
C-T19_1024	19	19456	42	43008	7.91	8.42	41.46	147.35
C-T40_512	32	16384	42	21504	3.45	5.62	20.62	144.43
C-T64_1024	15	15360	33	33792	4.62	4.95	24.67	68.20
C-T144_512	20	10204	31	15872	2.05	2.62	18.43	45.80
E-ami49L21_128	112	14366	196	25088	21.08	29.02	230.56	2997.43
E-ami49LT21_128	108	13824	196	25088	19.67	24.61	158.74	2442.70
E-TMCNCGSRC_064	204	13056	305	19456	26.11	35.73	346.61	6609.45
E-B10_512	36	18432	56	28672	5.64	7.65	54.62	249.38
E-B30_128	116	14848	236	30208	28.47	38.62	367.72	3450.55
E-T19_256	76	19456	168	43008	27.97	36.52	400.45	2347.43
E-T40_128	128	16384	168	21504	15.38	21.90	130.73	2413.09
E-T64_256	60	15360	132	33792	15.44	21.07	257.26	1103.24
E-T144_128	80	10240	124	15872	7.00	9.09	110.62	628.38

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 7: Computation time for instances in class C-TMCNCGSRC

Instance	W	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
C-TMCNCGSRC_001	2376	51	51	76	76	0.01	0.01	0.07	0.56
C-TMCNCGSRC_002	3360	51	102	76	152	0.02	0.03	0.15	1.35
C-TMCNCGSRC_004	4752	51	204	76	304	0.06	0.06	0.38	3.09
C-TMCNCGSRC_008	6720	51	408	76	608	0.11	0.14	0.82	6.35
C-TMCNCGSRC_016	9504	51	816	76	1216	0.26	0.36	2.00	16.24
C-TMCNCGSRC_032	13441	51	1632	76	2432	0.60	0.88	4.78	40.51
C-TMCNCGSRC_064	19009	51	3264	76	4864	1.34	1.86	9.72	90.77
C-TMCNCGSRC_128	26882	51	6528	76	9728	2.91	3.90	19.46	178.45
C-TMCNCGSRC_256	38018	51	13056	76	19456	6.23	8.91	45.64	380.30

Table 8: Computation Time for instances in class E-TMCNCGSRC

Instance	W	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
E-TMCNCGSRC_001	21906	204	204	304	304	0.20	0.26	1.64	48.72
E-TMCNCGSRC_002	30981	204	408	304	608	0.46	0.68	3.74	113.68
E-TMCNCGSRC_004	43813	204	816	304	1216	1.03	1.59	8.56	253.87
E-TMCNCGSRC_008	61962	204	1632	304	2432	2.25	3.56	20.62	610.46
E-TMCNCGSRC_016	87627	204	3264	304	4864	5.51	8.67	43.73	1523.50
E-TMCNCGSRC_032	123924	204	6528	304	9728	12.31	18.80	90.63	3248.41
E-TMCNCGSRC_064	175255	204	13056	304	19456	26.11	35.73	346.61	6609.45

Figure 6 shows three example layouts obtained by the bottom-left, the best-fit and the PBF algorithms. These are layouts obtained for the instance named E-TMCNCGSRC_001. The occupation rate obtained by the bottom-left algorithm is 90.59%, that obtained by the best-fit algorithm is 79.51%, and that obtained by the PBF algorithm is 92.45%. The running time spent for these layouts are reported in Table 8.

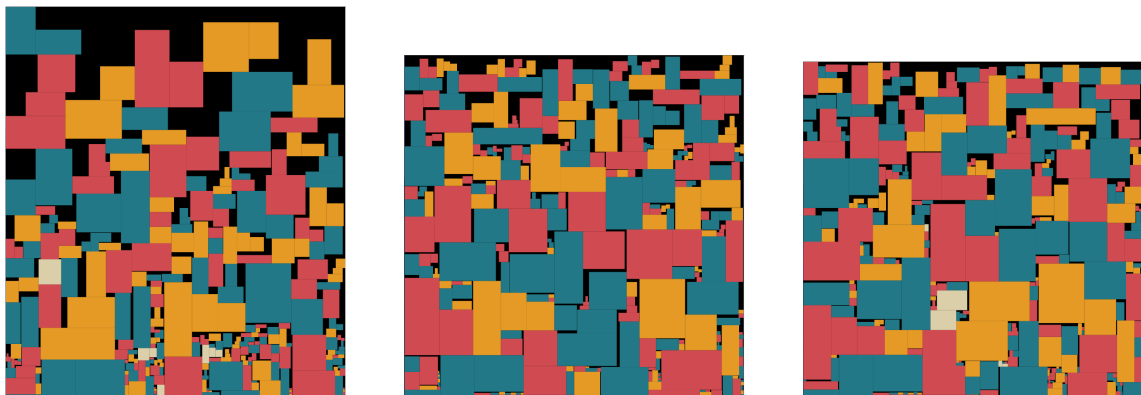

Figure 6: Layouts obtained for E-TMCNCGSRC_001 by the three algorithms (left: the best-fit algorithm, middle: the bottom-left algorithm, right: the PBF algorithm)

6. Conclusions

In this paper, we proposed a new constructive heuristic algorithm, the partition-based best-fit (PBF) algorithm, for the rectilinear block packing problem. The PBF algorithm can be regarded as a bridge between the bottom-left and the best-fit algorithms and takes advantages of both of these two algorithms. We analyzed the time complexity of the proposed PBF algorithm and showed that it runs in $O(mM \log M)$ time.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

We also proposed some effective rules utilized in the PBF algorithm and performed a series of experiments on 168 instances that were generated from nine benchmark instances. The occupation rate of the packing layouts obtained by the proposed PBF algorithm was more than 93% on average for these instances. The computational results show that the improvement on the performance of the occupation rate obtained by the PBF algorithm is remarkable compared with the bottom-left and the best-fit algorithms, and the PBF algorithm is especially effective for instances with many different sizes of shapes.

References

[1] R.C. Art: An approach to the two dimensional irregular cutting stock problem. IBM Cambridge Science Center, 36.Y08, 1966.

[2] B.S. Baker, E.G. Coffman Jr., and R.L. Rivest: Orthogonal packings in two dimensions. *SIAM Journal on Computing*, **9** (1980), 846–855.

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

[4] D. Chen, J. Liu, Y. Fu, and M. Shang: An efficient heuristic algorithm for arbitrary shaped rectilinear block packing problem. *Computers & Operations Research*, **37** (2010), 1068–1074.

[5] K. Fujiyoshi and H. Murata: Arbitrary convex and concave rectilinear block packing using sequence-pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **19** (2000), 224–233.

[6] Y. Hu, H. Hashimoto, S. Imahori, and M. Yagiura: Efficient implementations of construction heuristics for the rectilinear block packing problem. *Computer & Operations Research*, **53** (2015), 206–222.

[7] S. Jakobs: On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, **88** (1996), 165–181.

[8] M. Kang and W.W.-M. Dai: General floorplanning with L-shaped, T-shaped and soft blocks based on bounded slicing grid structure. *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, (1997), 265–270.

[9] M.Z. Kang and W.W.-M. Dai: Arbitrary rectilinear block packing based on sequence pair. *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, (1998), 259–266.

[10] C. Kodama, K. Fujiyoshi, and A. Ikeda: A fast algorithm for rectilinear block packing based on selected sequence-pair. *Integration, the VLSI Journal*, **40** (2007), 274–284.

[11] J.M. Lin, H.L. Chen, and Y.W. Cheng: Arbitrarily shaped rectilinear module placement using the transitive closure graph representation. *IEEE Transactions on VLSI Systems*, **10** (2002), 886–901.

[12] Y. Ma, X. Hong, S. Dong, Y. Cai, C.-K. Cheng and J. Gu: Stairway compaction using corner block list and its applications with rectilinear blocks. *ACM Transactions on Design Automation of Electronic System*, **9** (2004), 199–211.

[13] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani: Module packing based on the BSG-structure and IC layout applications. *IEEE Transactions on CAD of Integrated Circuits and Systems*, **17** (1998), 519–530.

[14] Y. Pang, C.K. Cheng, K. Lampaert, and W. Xie: Rectilinear block packing using O-tree representation. *Proceedings of the 2001 International Symposium on Physical Design (ISPD)*, (2001), 156–161.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

[15] G. Wäscher, H. Haußner, and H. Schumann: An improved typology of cutting and packing problems. *European Journal of Operational Research*, **183** (2007), 1109–1130.

[16] G.M. Wu, Y.C. Chang, and Y.W. Chang: Rectilinear block placement using B*-trees. *ACM Transactions on Design Automation of Electronic Systems*, **8** (2003), 188–202.

[17] J. Xu, P.N. Guo, and C.K. Cheng: Rectilinear block placement using sequence-pair. *Proceedings of the 1998 International Symposium on Physical Design (ISPD)*, (1998), 173–178.

Appendix A. Details of the Computational Results

We address in Table 9 to 12 the details of the computational results for each class of instances by the PBF algorithm when it uses one of the rules from FirstGrp, LastGrp, LargeGrp and BigGapGrp to choose a group to divide. The BestOccup of every instance is the best occupation rate obtained by the PBF algorithm with a specified pair of rules for choosing and partitioning groups. As explained in Section 5, the avgOccup is defined as the average value of BestOccup over all instances in one class. The columns of *BndArea*, *Area*, *Width* and *Height* are the avgOccup in percent obtained by the PBF algorithm with the Size-based rules based on the values of the bounding areas, areas, widths and heights of items. The columns of BL-midway and BL-final are the avgOccup in percent obtained with the adaptive rules BL-midway and BL-final. We choose the decreasing order of area as the priority among the items when the adaptive rules are utilized. When the Size-based rules are used, the priority among the items is set to the same criterion as the one used to partition a group.

Table 9: Computational results using the rule FirstGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.54	89.09	89.04	89.28	*89.61	89.56
C-ami49LT21	87.83	87.10	*89.67	88.99	88.68	89.38	89.48	89.52
C-TMCNCGSRC	86.92	81.71	89.08	89.59	87.56	87.61	89.49	*89.70
C-B10	87.66	90.32	90.36	91.28	89.95	90.82	91.28	*91.68
C-B30	84.52	88.14	88.17	88.50	88.63	88.42	*89.16	88.91
C-T19	73.79	76.24	76.60	77.16	77.66	75.62	*77.68	77.32
C-T40	94.09	91.24	95.89	95.95	94.47	*96.02	94.60	94.88
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	96.14	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.18	94.96	94.43	94.78	*95.51	95.42
E-ami49LT21	94.72	87.66	94.96	95.02	94.14	95.10	*95.40	95.07
E-TMCNCGSRC	92.89	83.55	*93.40	92.95	91.51	92.30	93.31	93.16
E-B10	93.93	87.92	94.72	94.42	94.94	94.30	*95.45	95.36
E-B30	94.58	89.20	94.64	95.04	94.55	93.83	*95.33	95.33
E-T19	86.95	84.19	87.38	*88.61	86.06	87.76	88.27	87.88
E-T40	98.01	90.63	*98.74	98.60	97.58	98.68	98.42	98.60
E-T64	95.33	92.87	95.52	95.93	95.04	95.73	*96.33	95.96
E-T144	97.83	96.71	*98.72	98.14	98.31	98.33	98.12	97.82

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 10: Computational results using the rule LastGrp to choose a group

Instance	BL	BF	Size-based				Adaptive	
			BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.48	89.15	89.08	89.22	89.57	*89.68
C-ami49LT21	87.83	87.10	89.29	89.10	88.74	88.98	89.51	*89.57
C-TMCNCGSRC	86.92	81.71	*89.92	89.58	87.46	87.98	89.18	88.96
C-B10	87.66	90.32	90.36	91.28	90.25	91.15	*91.35	*91.35
C-B30	84.52	88.14	87.88	88.50	88.54	88.42	88.52	*88.58
C-T19	73.79	76.24	76.51	77.14	77.56	75.22	*77.68	77.15
C-T40	94.09	91.24	95.67	95.98	94.87	*96.20	94.70	94.69
C-T64	86.99	92.92	*93.43	92.92	93.24	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.31	95.21	95.30	95.24	94.40
E-ami49L21	94.71	89.51	95.27	*95.48	95.02	95.17	95.33	95.47
E-ami49LT21	94.72	87.66	94.91	95.29	95.00	95.18	*95.30	95.08
E-TMCNCGSRC	92.89	83.55	*93.88	93.78	92.15	92.76	93.48	93.35
E-B10	93.93	87.92	95.58	95.33	95.06	*96.33	95.76	95.64
E-B30	94.58	89.20	95.02	95.31	94.63	94.72	95.22	*95.37
E-T19	86.95	84.19	87.42	89.08	87.35	88.52	*89.09	88.55
E-T40	98.01	90.63	*98.77	98.63	97.61	98.76	97.81	98.08
E-T64	95.33	92.87	96.38	96.34	96.42	*97.34	96.45	96.41
E-T144	97.83	96.71	*98.72	98.00	98.14	98.58	98.15	97.81

Table 11: Computational results using the rule LargeGrp to choose a group

Instance	BL	BF	Size-based				Adaptive	
			BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	*89.48	89.04	89.07	89.37	89.45	89.40
C-ami49LT21	87.83	87.10	89.34	89.12	88.69	89.41	*89.75	89.62
C-TMCNCGSRC	86.92	81.71	*89.82	89.60	87.49	88.09	89.47	89.51
C-B10	87.66	90.32	90.36	91.28	90.25	90.82	90.96	*91.35
C-B30	84.52	88.14	87.88	88.50	88.54	88.42	*88.68	*88.68
C-T19	73.79	76.24	76.51	77.14	77.66	75.51	*77.68	77.15
C-T40	94.09	91.24	95.61	95.87	94.87	*96.20	94.84	94.71
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.31	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.34	95.43	95.02	95.18	95.42	*95.49
E-ami49LT21	94.72	87.66	95.07	95.24	94.79	95.26	95.47	*95.52
E-TMCNCGSRC	92.89	83.55	*94.08	93.77	92.15	92.63	93.55	93.31
E-B10	93.93	87.92	95.43	95.34	94.99	*96.51	95.77	95.44
E-B30	94.58	89.20	95.17	95.20	94.69	94.65	95.39	*95.42
E-T19	86.95	84.19	87.34	*89.21	86.90	88.30	89.08	88.96
E-T40	98.01	90.63	98.84	98.60	97.60	*98.92	98.08	98.11
E-T64	95.33	92.87	96.89	96.45	96.38	*97.34	96.37	96.41
E-T144	97.83	96.71	98.72	98.43	98.14	*98.82	98.10	97.80

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 12: Computational results using the rule BigGapGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.47	89.44	88.98	89.22	89.59	*89.77
C-ami49LT21	87.83	87.10	89.57	89.72	88.79	89.30	89.69	*89.78
C-TMCNCGSRC	86.92	81.71	*89.50	88.61	87.42	88.03	89.49	89.40
C-B10	87.66	90.32	90.36	91.28	90.25	90.82	*91.35	*91.35
C-B30	84.52	88.14	88.30	*88.81	88.54	88.42	88.68	88.68
C-T19	73.79	76.24	76.51	76.85	77.48	75.55	*77.77	77.15
C-T40	94.09	91.24	96.06	94.90	94.49	*96.25	95.13	94.62
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.12	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.17	95.13	94.53	95.04	*95.60	95.43
E-ami49LT21	94.72	87.66	95.20	95.02	94.28	95.14	*95.53	95.44
E-TMCNCGSRC	92.89	83.55	93.50	93.11	91.81	92.79	*93.72	93.34
E-B10	93.93	87.92	95.45	94.39	94.99	*96.51	95.72	95.67
E-B30	94.58	89.20	95.33	95.24	94.78	94.82	*95.48	95.44
E-T19	86.95	84.19	87.37	88.71	86.85	88.25	88.88	*88.98
E-T40	98.01	90.63	98.74	98.64	97.62	*98.84	98.41	98.31
E-T64	95.33	92.87	96.69	96.45	96.38	*97.41	96.27	96.27
E-T144	97.83	96.71	*98.72	98.51	98.14	98.42	98.06	97.82

Yannan Hu
Graduate School of Information Science
Nagoya University
Furocho, Chikusaku
Nagoya 464-8601, Japan
E-mail: yannanhu@nagoya-u.jp

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.