EFFICIENT OVERLAP DETECTION AND CONSTRUCTION ALGORITHMS FOR THE BITMAP SHAPE PACKING PROBLEM

Yannan Hu
Nagoya University

Sho Fukatsu
Nagoya University

Hideki Hashimoto
Tokyo University of Marine Science and Technology

Shinji Imahori
Chuo University

Mutsunori Yagiura
Nagoya University

(Received February 3, 2017; Revised September 25, 2017)

Abstract The two-dimensional strip packing problem arises in wide variety of industrial applications. In this paper, we focus on the bitmap shape packing problem in which a set of arbitrarily shaped objects represented in bitmap format should be packed into a larger rectangular container without overlap. The complex geometry of bitmap shapes and the large amount of data to be processed make it difficult to check for overlaps. In this paper, we propose an efficient method for checking for overlaps and design efficient implementations of two construction algorithms, which are based on the bottom-left strategy. In this strategy, starting from an empty layout, items are packed into the container one by one. Each item is placed in the lowest position where there is no overlap relative to the current layout. We consider two algorithms, the bottom-left and the best-fit algorithm, which adopt this strategy. The computational results for a series of instances that are generated from well-known benchmark instances show that the proposed algorithms obtain good solutions in remarkably short time and are especially effective for large-scale instances.

Keywords: Combinatorial optimization, construction heuristics, strip packing, bitmap shapes, efficient implementation

1. Introduction

Two-dimensional strip packing problems are classical complex combinatorial optimization problems that arise in a wide variety of industrial applications, such as garment patterns, paper cutting and VLSI design. Many different types of cutting and packing problems have been studied in the literature [3, 7, 11, 15, 25]. Among them, problems of packing rectangles and those of irregular shapes have been extensively studied, and many exact and heuristic algorithms have been proposed [5, 8, 14, 16, 18, 19, 24].

In computer graphics, a bitmap image is a dot matrix data structure representing a grid of pixels. We assume that each pixel is square, and each pair of (horizontally or vertically) adjacent pixels share a common boundary. A bitmap shape is defined as the union of the interiors and boundaries of a set of pixels (see Figure 1(c)). Most computer images are stored in a bitmap format, such as GIF, JPEG, and PNG, which are popular on the World Wide Web. Any arbitrarily shaped object in a scene or image, no matter how complex its shape, can be approximately represented in bitmap format as shown in Figure 1.

In this paper, we focus on the problem of packing a set of bitmap shapes into a larger rectangular container with fixed width so as to minimize its height. A bitmap shape is technically characterized by the resolution of the image in pixels, and the computational
cost depends on the number of pixels. To reduce this cost, we design sophisticated data structures to efficiently deal with the large amount of data in bitmap shapes. Our basic idea is that we merge the pixels of a bitmap shape into rectangles so as to minimize the number of rectangles. This problem is known to be NP-hard [6]. We propose two methods: one uses a scan-line technique and the other solves this problem as a set covering problem.

Finally, a bitmap shape is represented by the union of these rectangles, whose relative positions are fixed. This implies that a bitmap shape can be treated as a rectilinear block, which is a polygonal block whose interior angles are either $90^\circ$ or $270^\circ$. Several heuristic methods have been proposed for the rectilinear block packing problem; these methods are based on various data structures to represent the relationships among the blocks: for example, BSG (bounded sliceline grid) [17, 22], sequence-pairs [9], O-tree [23], $B^*$-tree [26], TCG (transitive closure graph) [20], and CBL (corner block list) [21]. Hu et al. [13] generalized two construction algorithms for rectangle packing (the bottom-left algorithm [2] and the best-fit algorithm [4], which are known as the most remarkable of existing construction heuristics for the rectangle packing problem) to the case of the rectilinear block packing problem and gave efficient implementations for these two algorithms. They also proposed another heuristic algorithm in [12], which takes advantages of both the bottom-left and the best-fit algorithms.

In this paper, we design efficient implementations of two construction heuristics for the bitmap shape packing problem, the bottom-left and best-fit algorithms. The main strategy of these two algorithms is the bottom-left strategy, which is derived from the bottom-left algorithm for rectangle packing [2]. In this strategy, whenever a new item is being packed into the container, it will be placed at the bottom-left position (abbreviated as the BL position) relative to the current layout. This 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 can be moved neither left nor down.

In the implementations proposed in [13], a technique called the no-fit polygon (NFP) is used to check for overlaps among rectilinear blocks; the running time of the construction algorithms depends on the complexity of the NFPs, where the complexity of an NFP is defined to be the number of rectangles whose union represents the NFP. Note that considering the NFP of any pair of rectilinear blocks (bitmap shapes) as a set of rectangles, our algorithms can deal with two-dimensional objects having holes or separated parts. In this paper, we design a method to reduce the complexity of NFPs. As a consequence, we can efficiently check for overlaps among bitmap shapes. We then analyze the time complexity of the resulting implementations of the two construction algorithms and show that under a weak assumption, which is satisfied unless the shapes of given items are pathologically complex, their computation times grow almost linearly to the vertical or horizontal resolution of given shapes.

The computational results for a series of well-known benchmark instances show that the
proposed algorithms obtain good solutions in remarkably short times and are effective for large-scale instances.

2. Problem Description

We are given a set of \( n \) bitmap items \( \mathcal{B} = \{ B_1, B_2, \ldots, B_n \} \), where each item has a deterministic configuration and size taken 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 each of its four sides parallel to the \( x \)- or \( y \)-axis. The objective is to minimize the height \( H \) of the container used in packing all the given items. Note that minimization of the height \( H \) is equivalent to maximization of the occupation rate, defined as \( \sum_{i=1}^{n} A(B_i)/WH \), where \( A(B_i) \) denotes the area of bitmap item \( B_i \).

**Figure 2**: An example of the bitmap shape packing problem and a solution

Figure 2 shows an example of the bitmap shape packing problem. The number of bitmap items \( n \) is 7, and the number of different shapes \( t \) is 5. The task is to pack the seven given items (Figure 2(b)) into the rectangular container (Figure 2(a)) so as to minimize the height of the container. Figure 2(c) is an example packing layout after packing all the bitmap items into the container.

We define the bounding box of an item \( B_i \) as the smallest rectangle that encloses \( B_i \), and its width and height are denoted by \( w_i \) and \( h_i \). The location of an item \( B_i \) is described by the coordinates \((x_i, y_i)\) of its reference point, where the reference point is the bottom-left corner of its bounding box. For convenience, each bitmap item and the container \( C \) are regarded as sets of points (including both interior and boundary points), whose coordinates are given relative to the origin \( O = (0, 0) \). Then, we describe the bitmap item \( B_i \) placed at \( v_i = (x_i, y_i) \) by the Minkowski sum \( B_i \oplus v_i = \{ p + v_i \mid p \in B_i \} \). Let \( \text{int}(B_i) \) be the interior of \( B_i \). Then the bitmap shape packing problem is formally described as follows.

**Minimize** \( H \)

subject to

\[
\begin{align*}
0 \leq x_i & \leq W - w_i, & 1 \leq i \leq n & \quad (2.1) \\
0 \leq y_i & \leq H - h_i, & 1 \leq i \leq n & \quad (2.2) \\
\text{int}(B_i \oplus v_i) \cap (B_j \oplus v_j) & = \emptyset, & i \neq j. & \quad (2.3)
\end{align*}
\]

Constraints (2.1) and (2.2) require that all the bitmap items should be packed inside the container. Constraint (2.3) ensures that every item \( B_j \) does not have common points with the interior points of any other item \( B_i \), which requires that no item should overlap another.

For simplicity, the rotation and reflection of items are not considered in this work unless otherwise stated, because the results can be easily generalized to cases where such operations are allowed.
3. Basic Knowledge

In this section, we explain some important techniques and definitions used in our algorithms. A crucial technique for packing problems, the concept of a no-fit polygon, is explained in Section 3.1. The definition of the BL position in general and the FindBL algorithm to calculate the BL positions for rectilinear blocks [13] are introduced in Section 3.2 and Section 3.3, respectively. Two heuristic construction algorithms for the rectilinear block packing problem are then introduced in Section 3.4. We first explain the bottom-left algorithm in Section 3.4.1, which is one of the simplest algorithms based on the bottom-left strategy. Then we explain the best-fit algorithm in Section 3.4.2; this algorithm is slightly more complicated than the bottom-left algorithm. Finally, we report the time complexities of these algorithms in Section 3.4.3.

3.1. No-fit polygon

The no-fit polygon (NFP), which was introduced by Art [1] in the 1960s, is a geometric technique to check for overlaps of two polygons in two-dimensional space. This concept uses the term “shape envelope” to describe the positions in which two polygons can be placed without overlap. Shape envelopes are defined for an ordered pair of two polygons $P_i$ and $P_j$, where the position of polygon $P_i$ is fixed and polygon $P_j$ can be moved. We denote the NFP of $P_j$ relative to $P_i$ by $NFP(P_i, P_j)$. $NFP(P_i, P_j)$ is the set of positions of polygon $P_j$ having an intersection with polygon $P_i$, which is formally defined as

$$NFP(P_i, P_j) = \text{int}(P_i) \oplus (-\text{int}(P_j)) = \{u - w \mid u \in \text{int}(P_i), w \in \text{int}(P_j)\}.$$  

When the two relevant polygons are clear from the context, we may simply use $NFP$ instead of $NFP(P_i, P_j)$. Assume that $\partial NFP(P_i, P_j)$ denotes the boundary of $NFP(P_i, P_j)$, and $\text{cl}(NFP(P_i, P_j))$ denotes the closure of $NFP(P_i, P_j)$. The no-fit polygon has the following important properties, where $v_i$ and $v_j$ are the positions of $P_i$ and $P_j$:

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

Hence, the problem of checking whether two polygons overlap becomes an easier problem of checking whether a point is in a polygon.

![Figure 3: NFP of two rectangles](image)

When $P_i$ and $P_j$ are both rectangles, $NFP(P_i, P_j)$ is the interior of a rectangle, and it is not hard to show that the NFP of bitmap shapes can be represented as the union of (the interiors of) a set of rectangles. The basic idea of computing NFP of rectangles is illustrated by the example in Figure 3. When we are given two rectangles $R$ and $R'$, where rectangle $R$ (resp., $R'$) has width $w$ (resp., $w'$) and height $h$ (resp., $h'$), $NFP(R, R')$ can be computed by

$$NFP(R, R') = \{(x, y) \mid -w' < x < w, -h' < y < h\}.$$  

(3.1)
Note that $NFP(R, R')$ is an open rectangle, and it can be computed in $O(1)$ time.

In our heuristics, each bitamp shape is treated as a rectilinear block that is represented by a set of rectangles. Hence, it is not hard to compute the NFP of bitmap shapes and the details are given in Section 4.1.

### 3.2. Bottom-left position

Bottom-left stable feasible positions are defined for a given area, a set of bitmap items placed in the area, and one new item to be placed. “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. Thus, a bottom-left stable feasible position is a point in the area where new item can be placed without overlapping with already placed bitmap items and the new item cannot be moved left or down.

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 concepts of bottom-left stable feasible positions and the BL position are illustrated in Figure 4.

![Figure 4: Bottom-left stable feasible positions and the BL position](image)

### 3.3. Method to calculate the BL position for a bitmap shape

Our method to calculate the BL position for a bitmap shape is based on an algorithm called $FindBL$, which was proposed in [13]. The $FindBL$ algorithm was devised to calculate the BL position for a new rectilinear block to be placed relative to a rectangular container and a rectilinear block already placed in the container. The main idea of the $FindBL$ algorithm is to find bottom-left stable feasible positions by using the NFP technique and a sweep-line method. A sweep line is a line, parallel to the $x$-axis, that moves upward from the bottom of the container.

Assume that the NFPs of the new rectilinear block relative to all the items in the container are given and that this information is saved in a 2-3 tree. The BL position is the leftmost of the lowest positions in the container and having overlap number zero, where the overlap number of a point $v = (x, y)$ is the number of NFPs that contain $v$. The $FindBL$ algorithm also stores the overlap number of arbitrary points on the sweep line in the same 2-3 tree that maintains the information about NFPs. With the sweep line moving upward from the bottom, the bottom-left position appears as the leftmost point among the initially emerging points on the sweep line whose overlap numbers are 0.

Considering that a bitmap shape can be treated as a rectilinear block, we can naturally use this $FindBL$ algorithm to calculate BL positions for the bitmap items. Assuming that...
3.4. Heuristic construction algorithms for rectilinear block packing

In this section, we introduce two heuristic construction algorithms for the rectilinear block packing problem. With the implementation in [13], both of these algorithms use the FindBL algorithm to calculate BL positions for the rectilinear blocks.

3.4.1. Bottom-left algorithm for rectilinear block packing

In this section, we explain the bottom-left algorithm for the rectilinear block packing problem. The basic idea of this algorithm comes from the bottom-left algorithm for the rectangle packing problem, which was proposed by Baker et al. [2].

The bottom-left algorithm can be explained as follows. We are given a set of $n$ rectilinear blocks $S = \{S_1, S_2, \ldots, S_n\}$ and an order of those 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 in the current layout (i.e., the layout just before it is placed).

3.4.2. 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 this algorithm comes from the best-fit algorithm for the rectangle packing problem, which was proposed by Burke et al. [4].

The best-fit algorithm can be explained as follows. We are given a set of $n$ rectilinear blocks $S = \{S_1, S_2, \ldots, S_n\}$ with a specified priority (e.g., an item with a larger area 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 BL positions of all of the remaining items relative to the current layout. In each iteration, the rectilinear block whose BL position has 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.

3.4.3. Time complexities

In this section, we derive the time complexities of the bottom-left and best-fit algorithms for rectilinear block packing problems.

Assume that the number of rectangles that represent a rectilinear block $S_j$ is denoted by $\kappa_j$, and let $K = \sum_{j=1}^{n} \kappa_j$. We also assume that each rectilinear block takes a deterministic shape from a set of $d$ shapes $D = \{D_1, D_2, \ldots, D_d\}$, and when $d < n$, some items have identical shapes. We define $\kappa_i^D$ to be the number of rectangles that represent shape $D_i$, and let $\kappa = \sum_{i=1}^{d} \kappa_i^D$.

Next we briefly explain the basic idea of the efficient implementation of the bottom-left and best-fit algorithms in [13]. They used the no-fit polygon technique to check for overlaps among items and to compute the BL positions of items. The NFP of an item $S_j$ relative to $S_i$ has the following property: the reference point of $S_j$ is contained in the NFP if and only if $S_j$ overlaps with $S_i$. When the algorithm computes the BL position of an item $S_j$, it uses the NFPs of $S_j$ relative to the items in the container, and these NFPs are placed at the positions where the corresponding items are placed. They call such a layout of NFPs an NFP layout for $S_j$. One of the advantages of such a layout is that the problem of finding the BL position of $S_j$ 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 $S_j$ and $S_{j'}$ are the same, their NFP layouts are the same. Hence, they only need to keep $d = |D|$ NFP layouts, one for each distinct shape in $D$, 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 heuristic. The basic idea is to dynamically store the NFP layout with respect to the current packing layout for each shape in $D$ during the packing process. Whenever an item is to be placed into the container, the algorithm computes the BL position of every shape $D_j$ by using the NFP layout for $D_j$. It then chooses an item $S_i$ to place in this iteration (e.g., in the case of the best-fit algorithm, the item whose BL position is the leftmost among the lowest) and places it at its BL position. The algorithm then updates the NFP layout of every shape $D_j$ in $D$, adding to the NFP layout the NFP of $D_j$ relative to $S_i$. It is shown in [13] that throughout the entire computation of the bottom-left and the best-fit algorithm, the total running time for computing BL positions of shape $D_j$ is $O(\kappa_j K \log K)$, including the time to update the NFP layout of $D_j$. The total computation time of this part for all shapes is therefore $\sum_{j=1}^{d} O(\kappa_j D_j K \log K) = O(\kappa K \log K)$. This dominates the running time of the other parts of the algorithm. As a consequence, both of the bottom-left and the best-fit algorithm run in $O(\kappa K \log K)$ time.

4. Heuristic Algorithms for the Bitmap Shape Packing Problem

In this section, we explain the bottom-left and the best-fit algorithms for the bitmap shape packing problem, which are generalized from the algorithms for rectilinear block packing. We first introduce a simple method to calculate NFPs for bitmap shapes in Section 4.1. After that, we generalize the two heuristic construction algorithms to the case of the bitmap shape packing problem in Section 4.2. Finally, we report the time complexities of our algorithms in Section 4.2.3.

4.1. Method to calculate NFPs for bitmap shapes

Assume that each bitmap shape is represented as a set of rectangles whose relative positions are fixed, and let $m_i$ be the number of rectangles that represent bitmap item $B_i$ ($m_i$ is not necessarily the same as (and is usually much smaller than) the number of pixels in $B_i$, as will be discussed in Section 5). We also assume that each such rectangle has a positive area; in other words, special cases of rectangles such as line segments and points are not considered. Note that there is no restriction on the way the relative positions are fixed; for example, an item $B_i$ can be any set of separate rectangles as long as their relative positions are fixed.

We now consider the case when two bitmap items $B_i$ and $B_j$ are given. Let $\mathcal{R}_i = \{R_{i1}, R_{i2}, \ldots, R_{im_i}\}$ be the set of rectangles that represents $B_i$, and $v_{ik}$ be the position of $R_{ik}$ relative to the reference point of $B_i$ for $k = 1, 2, \ldots, m_i$, that is,

$$B_i = \bigcup_{k=1}^{m_i} (R_{ik} \oplus v_{ik}).$$

The set $\mathcal{R}_j$ is defined similarly. Then $\text{NFP}(B_i, B_j)$ is the union of $\text{NFP}(R_{ik}, R_{jl})$ for all pairs of $R_{ik}$ and $R_{jl}$. That is, $\text{NFP}(B_i, B_j)$ is formally given by

$$\text{NFP}(B_i, B_j) = \bigcup_{k=1}^{m_i} \bigcup_{l=1}^{m_j} (\text{NFP}(R_{ik}, R_{jl}) \oplus v_{ik} \oplus (-v_{jl})). \quad (4.1)$$
For each rectangle $R_{ij}$, we can easily calculate its NFP with respect to $R_{ik}$ by using (3.1). Hence $NFP(B_i, B_j)$ consists of $m_i m_j$ rectangles, and it can be computed in $O(m_i m_j)$ time. For convenience, we call such rectangles NFP rectangles and we define the complexity of an NFP as the number of NFP rectangles.

4.2. Heuristic algorithms for the bitmap shape packing problem

We now generalize the two heuristic algorithms, the bottom-left and the best-fit algorithm, that we introduced in Section 3.4 to the case of the bitmap shape packing problem. We calculate NFPs for the bitmap shape by the method explained in Section 4.1, and we use the FindBL algorithm explained in Section 3.3 to compute the BL positions for bitmap shapes.

4.2.1. Bottom-left algorithm for the bitmap shape packing problem

The bottom-left algorithm can be explained as follows. We are given a set of $n$ bitmap shapes $B$ and an order of those 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 for the current layout (i.e., the layout just before it is placed).

The bottom-left algorithm for the bitmap shape packing problem, in which FindBL algorithm is used to find BL positions, is formally described in Algorithm 1, where we assume for simplicity that $\{B_1, B_2, \ldots, B_n\}$ are packed in order of increasing index.

Algorithm 1 Bottom-left algorithm

1: Set $j := 0$.
2: Set $j := j + 1$. If $j > n$, output the packing layout and stop.
3: By using the FindBL algorithm, find the BL position $(x, y)$ of $B_j$.
4: Pack $B_j$ at $(x, y)$.
5: For each shape $T_i$ ($i = 1, 2, \ldots, t$), update the NFP layout for $T_i$ by inserting $NFP(B_j, T_i)$. Then return to Step 2.

4.2.2. Best-fit algorithm for the bitmap shape packing problem

The best-fit algorithm can be explained as follows. We are given a set of $n$ bitmap items $B$ with specified priorities (e.g., items with larger area have higher priority). The algorithm packs items one by one and, in each iteration, it dynamically chooses a bitmap item to pack from the remaining items by the following rule. First calculate the BL positions of all the remaining items. Then the bitmap item whose BL position has the smallest $x$-coordinate among those with the lowest $y$-coordinate is packed in this iteration. If more than one such item exists, the one with the highest priority is chosen. The best-fit algorithm for the bitmap shape packing problem is formally described in Algorithm 2.

4.2.3. Time complexities

In this section, we give the time complexities of Algorithms 1 and 2 for the bitmap shape packing problem.

Let $f_j$ be the number of pixels that represent a bitmap item $B_j$, and let $F = \sum_{j=1}^{n} f_j$. Each bitmap item has a deterministic configuration and size taken from the set of $t$ shapes $T = \{T_1, T_2, \ldots, T_t\}$, and when $t < n$, some shapes are repeated among the items. We define $f_i^T$ as the number of rectangles that represent shape $T_i$, and $f$ as the sum of $f_i^T$ for all of the shapes in $T$. By a similar argument to the one in Section 3.4.3, both the bottom-left and best-fit algorithms run in $O(f F \log F)$ time with the efficient implementations proposed in [13].

In general, $K$, the number of rectangles that represent a rectilinear block $S_j$, is much smaller than $F$, the number of pixels that represent a bitmap item $B_j$. This implies that
obtaining a solution for the bitmap shape packing problem costs much more time than the rectilinear packing problem. Also, the number of pixels of bitmap items increases when the resolution and the precision become better. This indicates that the computational cost increases when we deal with high-resolution objects. Indeed, if we treat a bitmap item as a set of pixels, we cannot obtain a solution in a reasonable time, even for instances with only several high-resolution items, because the computational cost is too high. In Section 5, we focus on this fact and propose two methods to represent bitmap items compactly so as to reduce the computational cost for high-resolution items.

Algorithm 2 Best-fit algorithm
1: Set $B' := B$.
2: If $B' = \emptyset$, output the current layout and stop.
3: Set $T' := T$, $x^* := +\infty$ and $y^* := +\infty$.
4: If $T' = \emptyset$, go to Step 7.
5: Choose a shape $T_i$ in $T'$, and then let $T' := T' \setminus \{T_i\}$. If there is no item in $B'$ whose shape is $T_i$, return to Step 4.
6: Find the BL position $(x,y)$ of $T_i$ by using the FindBL algorithm. Let $B_j \in B'$ be the item that takes the highest priority among those items whose shape is $T_i$. If one of the following three conditions holds, then let $x^* := x$, $y^* := y$ and $j^* := j$.
   (i) $y < y^*$.
   (ii) $y = y^*$ and $x < x^*$.
   (iii) $y = y^*$, $x = x^*$ and $B_j$ has higher priority than $B_{j^*}$.
   Return to Step 4.
7: Pack the item $B_{j^*}$ at $(x^*, y^*)$.
8: For each shape $T_i$ ($i = 1, 2, \ldots, t$), update the NFP layout for $T_i$ by inserting NFP($B_{j^*}, T_i$). Set $B' := B' \setminus \{B_{j^*}\}$ and return to Step 2.

5. Compact Representations for Bitmap Shapes
We propose two representation methods, called the Scan-line and Rectangle representations, to reduce the complexity of the representation of bitmap shapes. In both cases we merge as many pixels as possible into rectangles so as to minimize the number of rectangles representing a bitmap shape. This problem is known to be NP-hard (p. 232 of [10]).

5.1. Scan-line representation
We assume that each pixel has width one and height one. The Scan-line representation reduces the complexity of bitmap shapes by merging the pixels in each row into rectangles with height one and width as large as possible. Then each bitmap shape is represented as a rectilinear block consisting of a set of rectangles of height one. The Scan-line representation is illustrated in Figure 5. For the example in Figure 5, we are given the bitmap shape on the left, which is represented by 84 pixels. By using the Scan-line representation, we can represent this shape as 15 rectangles (see the image on the right of Figure 5).

We define a bitmap shape to be $y$-monotone if it can be represented by a set of rectangles such that there is exactly one rectangle for each row in the bounding box and rectangles in adjacent rows are contiguous (i.e., they share a line segment of positive length). Note that if a bitmap item $B_i$ is $y$-monotone, it can be represented by $h_i$ rectangles, where $h_i$ is the height of its bounding box. Figure 6 shows examples of $y$-monotone and non-$y$-monotone bitmap shapes. The item in the middle of Figure 6 is not a $y$-monotone shape because rectangles 3 and 4 are not contiguous. The item on the right of Figure 6 is also not a
y-monotone bitmap shape because there is more than one rectangle in a single row (the rectangles 3 and 4 are in the same row).

![Image](image.png)

Figure 5: An example of Scan-line representation

![Image](image.png)

Figure 6: Examples of y-monotone and non-y-monotone bitmap shapes

5.2. Rectangle representation

The Rectangle representation is more effective in reducing the number of rectangles than the Scan-line representation. In order to represent a bitmap shape, we formulate the problem of minimizing the number of rectangles used to represent a bitmap shape as a set covering problem (SCP), which is a classical combinatorial optimization problem. We are given a set of elements $E = \{1, 2, \ldots, e\}$ (called the universe) and a set $Q = \{Q_1, Q_2, \ldots, Q_u\}$, where $Q_j \subseteq E, \forall j \in \{1, 2, \ldots, u\}$, and $\bigcup_{j=1}^u Q_j = E$. The objective is to find a subset of $Q$ with the smallest size that covers all the elements in $E$.

In our problem, we take $E$ to be the set of all the pixels in $B_i$ and $Q$ as the set of all the rectangles generated by the following two rules:

- **Rule 1.** For each $y$-coordinate, we merge the pixels into rectangles of height one with width as large as possible. Then we increase the height of these rectangles as much as possible.
- **Rule 2.** For each $x$-coordinate, we merge the pixels into rectangles of width one with height as large as possible. Then we increase the width of these rectangles to their right as much as possible.

The corresponding SCP can be formulated as the following integer linear programming problem.

\[
\text{Minimize} \quad \sum_{j: Q_j \in Q} c_j x_j \quad (5.1)
\]

subject to

\[
\begin{align*}
\sum_{j: Q_j \in Q} a_{ij} x_j & \geq 1 \quad \forall i \in E \\
x_j & \in \{0, 1\} \quad \forall j : Q_j \in Q,
\end{align*}
\]

where $x_j$ is a 0-1 variable associated with the subset $Q_j$. If rectangle $j$ is selected, then $x_j = 1$, and otherwise $x_j = 0$. The binary coefficient $a_{ij} = 1$ when the subset $Q_j$ covers element $i$, otherwise, $a_{ij} = 0$. The cost $c_j$ equals 1 for all $j$, since our objective is to minimize
the number of rectangles. We solve the resulting SCP by using the SCP solver proposed in [27]. The Rectangle representation is illustrated in Figure 7. We generate 15 rectangles by Rule 1 and 14 rectangles by Rule 2 from a bitmap shape with 84 pixels. As a result the bitmap shape is represented by eight rectangles, which are obtained by the SCP solver.

Figure 7: An example of Rectangle representation

6. Efficient Implementation

In this section, we propose an efficient implementation, called procedure *Integrate-NFP*, to check for the overlapping of bitmap shapes. The main objective is to reduce the time complexities of both the bottom-left and the best-fit algorithms for the bitmap shape packing problem. The basic idea is that we reduce the complexity of NFPs and prepare a compact representation of the NFP for every pair of shapes in advance for use in the packing algorithms.

6.1. Procedure *Integrate-NFP* for checking for overlaps

Recall the analysis of the time complexities of the bottom-left and the best-fit algorithm for the bitmap shape packing problem discussed in Section 4.2.3. The running time of both the bottom-left and the best-fit algorithms depends on the complexity of NFPs, which was defined to be the number of rectangles that represent the NFPs. The number of rectangles that represent a bitmap item $B_i$ is denoted by $m_i$ as explained in Section 4.1, and we define $M = \sum_{i=1}^{n} m_i$. We also define $m_j^T$ as the number of rectangles that represent shape $T_j$, and $m = \sum_{i=1}^{t} m_j^T$ for all of the shapes in $T$. Both the bottom-left and the best-fit algorithm run in $\sum_{i=1}^{n} \sum_{j=1}^{t} O((m_j^T m_i) \log M) = O(M m \log M)$ time, where $m_j^T m_i$ is the complexity of the NFP of item $B_i$ relative to shape $T_j$.

Utilizing the Scan-line representation or the Rectangle representation, the complexity of NFPs is significantly reduced compared with the NFPs for the original representation of bitmap items. We design a procedure *Integrate-NFP* to further reduce the complexity of NFPs. The basic idea is to prepare a compact representation of the NFP for every pair of shapes $T_i$ and $T_j$ in $T$. When $T_i$ and $T_j$ are represented by the Scan-line representation, the height of each rectangle in the representations of $T_i$ and $T_j$ is one. This implies that if we calculate the NFP of $T_j$ relative to $T_i$ by the method explained in Section 4.1, then $NFP(T_i, T_j)$ is represented by a set of rectangles where the height of each rectangle is two. The complexity of NFPs can be reduced significantly by merging overlapping rectangles with identical $y$-coordinates into one large rectangle.

The procedure *Integrate-NFP* calculates the NFP for every pair of shapes and reduces the number of rectangles representing the NFP by repeatedly merging a pair of overlapping rectangles with the same $y$-coordinate into one until such merging is not possible. In other words, we prepare compact representations of $t \times t = t^2$ NFPs in advance.
The procedure Integrate-NFP is illustrated in Figure 8. In this example we are given two shapes, $T_i$ and $T_j$, and we compute $NFP(T_i, T_j)$. If we simply use expression (4.1) to compute the NFP, it takes 16 rectangles to represent $NFP(T_i, T_j)$. Applying the procedure Integrate-NFP, the number of rectangles required to represent $NFP(T_i, T_j)$ is reduced to seven.

![Figure 8: An example of procedure Integrate-NFP](image)

For the number of rectangles needed to represent an NFP when procedure Integrate-NFP is used, we have the following lemma.

**Lemma 6.1.** If both bitmap items $B_i$ (with height $h_i$) and $B_j$ (with height $h_j$) are y-monotone shapes, then $NFP(B_i, B_j)$ is also a y-monotone shape and can be represented by $h_i + h_j - 1$ rectangles.

**Proof.** We represent $B_i$ and $B_j$ as sets of rectangles with height one in the Scan-line representation. Let $\mathcal{R}_i = \{R_{i,1}, R_{i,2}, \ldots, R_{i,m_i}\}$ and $\mathcal{R}_j = \{R_{j,1}, R_{j,2}, \ldots, R_{j,m_j}\}$ be the sets of such rectangles representing $B_i$ and $B_j$, respectively. For convenience, we assume that the rectangles in $\mathcal{R}_i$ and $\mathcal{R}_j$ are numbered from top to bottom, such as with $R_{i,1}$ at the top, $R_{i,2}$ next to $R_{i,1}$, and $R_{i,m_i}$ at the bottom.

According to the method explained in Section 4.1, $NFP(B_i, B_j)$ is represented by a set of rectangles with height two and the height of $NFP(B_i, B_j)$ is $h_i + h_j$. To prove the lemma, we need to prove only that the shape of the resulting $NFP(B_i, B_j)$ is also y-monotone.

We consider any two consecutive rectangles $R_{i,p}$, $R_{i,p+1}$ in $\mathcal{R}_i$, and any two consecutive rectangles $R_{j,q}$, $R_{j,q+1}$ in $\mathcal{R}_j$. Assume that $B_i$ has already been packed into the container (i.e., the position of $B_i$ is fixed) and $B_j$ is the new item to be packed (i.e., $B_j$ is movable). As depicted in Figure 9, let $l_{i,p}$ be the $x$-coordinate of the left edge of $R_{i,p}$ measured from the origin (0, 0) of the container $C$, and let $r_{i,p}$ be the $x$-coordinate of the right edge of $R_{i,p}$. We define $l_{i,p+1}$ and $r_{i,p+1}$ similarly. Also let $d_{i,j,q}$ be the horizontal distance between the left edge of $R_{j,q}$ and the reference point of $B_j$, and let $d_{r,j,q}$ be the horizontal distance between the right edge of $R_{j,q}$ and the reference point of $B_j$. We define $d_{i,j,q+1}$ and $d_{r,j,q+1}$ similarly. Then the width of $R_{j,q}$ is $d_{r,j,q} - d_{l,j,q}$ and that of $R_{j,q+1}$ is $d_{r,j,q+1} - d_{l,j,q+1}$.

Let us consider $NFP(R_{i,p+1}, R_{j,q+1})$ and $NFP(R_{i,p}, R_{j,q})$ that are rectangles with height two. Let $\alpha$ be the $x$-coordinate of the left boundary and $\beta$ be that of the right boundary of $NFP(R_{i,p}, R_{j,q})$. Similarly, let $\gamma$ be the $x$-coordinate of the left boundary and $\delta$ be that of the right boundary of $NFP(R_{i,p+1}, R_{j,q+1})$ (Figure 10). Note that $NFP(R_{i,p+1}, R_{j,q+1})$ and

*Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.*
Figure 9: Two consecutive rectangles in \( R_i \) and \( R_j \)

\( NFP(R_{i,p}, R_{j,q}) \) have the same \( y \)-coordinate and they overlap if and only if \( \gamma \) is smaller than \( \beta \) and \( \alpha \) is smaller than \( \delta \). Because items \( B_i \) and \( B_j \) are \( y \)-monotone shapes, the following inequalities hold:

\[
\begin{align*}
    l_{i,p+1} &< r_{i,p} & (6.1) \\
    d_{i,j,q} &< d_{r_{j,q+1}} & (6.2) \\
    l_{i,p} &< r_{i,p+1} & (6.3) \\
    d_{l_{j,q+1}} &< d_{r_{j,p}} & (6.4)
\end{align*}
\]

Using inequalities (6.1) to (6.4), we have

\[
\begin{align*}
    l_{i,p+1} - d_{r_{j,q+1}} &< r_{i,p} - d_{i,j,q} & (6.5) \\
    l_{i,p} - d_{r_{j,q}} &< r_{i,p+1} - d_{i,j,q+1} & (6.6)
\end{align*}
\]

Because \( \gamma \) is equal to \( l_{i,p+1} - d_{r_{j,q+1}} \) and \( \beta \) is equal to \( r_{i,p} - d_{i,j,q} \), (6.5) implies that \( \gamma \) is smaller than \( \beta \). Similarly, since \( \alpha \) equals \( l_{i,p} - d_{r_{j,q}} \) and \( \delta \) equals \( r_{i,p+1} - d_{i,j,q+1} \), (6.6) implies that \( \alpha \) is smaller than \( \delta \). That is, every pair of \( NFP(R_{i,p}, R_{j,q}) \) and \( NFP(R_{i,p+1}, R_{j,q+1}) \) always overlap with one another.

Then, let us consider rectangles in \( R_i \) and \( R_j \) with the same \( y \)-coordinate. Assume without loss of generality that the \( y \)-coordinate of (the bottom edge of) \( R_{i,m_i} \) equals 0. Let \( z \) be an integer from \( 1 - m_i \) to \( m_j - 1 \), and consider all pairs of \( p \) and \( q \) that satisfy \( z = q - p \) (\( 1 - m_i \leq z \leq m_j - 1 \)). The NFP rectangles with the same \( y \)-coordinate are all \( NFP(R_{i,p}, R_{j,q}) \) that satisfy \( z = q - p \) (\( 1 \leq p \leq m_i \), \( 1 \leq q \leq m_j \)). Recall that we proved that any pair of \( NFP(R_{i,p}, R_{j,q}) \) and \( NFP(R_{i,p+1}, R_{j,q+1}) \) always overlap. If there is more than one NFP rectangle at one \( y \)-coordinate, every NFP rectangle \( NFP(R_{i,p}, R_{j,q}) \) overlaps with at least one of the rectangles \( NFP(R_{i,p+1}, R_{j,q+1}) \) and \( NFP(R_{i,p-1}, R_{j,q-1}) \). Hence, we can merge all NFP rectangles with the same \( y \)-coordinate into one rectangle.

Denoting by \( y_{NFP} \) the \( y \)-coordinate of the bottom edge of the lowest rectangle that represents \( NFP(B_i, B_j) \), that of the top rectangle equals \( y_{NFP} + h_i + h_j - 2 \). Therefore, the number of distinct \( y \)-coordinates of rectangles that represent \( NFP(B_i, B_j) \) is \( (y_{NFP} + h_i + h_j - 2) - y_{NFP} + 1 = h_i + h_j - 1 \). This indicates that the \( NFP(B_i, B_j) \) can be represented by \( h_i + h_j - 1 \) rectangles.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.
6.2. Time complexities of the construction algorithms

In this section, we analyze the time complexities of our algorithms for the bitmap shape packing problem with using procedure Integrate-NFP to represent NFPs compactly. Below, $m_i (m_j^T)$ is the number of rectangles that the Scan-line representation requires to represent a bitmap item $B_i$ (shape $T_j$).

Assume that all the shapes $T_j$ in $\mathcal{T}$ can be divided into at most $\xi$ parts such that each part is $y$-monotone and $\xi$ is a constant. This assumption is natural and is satisfied unless the given shape is extremely complex. Note that under this assumption, $m_i = O(h_i)$ and $m_j^T = O(h_j^T)$ hold for the heights $h_i$ of $B_i$ and $h_j^T$ of $T_j$; that is, $m_i$ and $m_j^T$ are linear to the vertical resolution. According to Lemma 6.1, $NFP(T_j, T_k)$ can be represented with at most $\xi^2(m_j^T + m_k^T) = O(m_j^T + m_k^T)$ rectangles of height two.

For every pair of shapes $T_j$ and $T_k$ in $\mathcal{T}$, $m_j^T m_k^T$ rectangles that represent $NFP(T_j, T_k)$ need to be merged. The time needed for preparing NFPs by the procedure Integrate-NFP for all the pairs in $\mathcal{T}$ is $\sum_{j=1}^{t} \sum_{k=1}^{t} m_j^T m_k^T = O(m^2)$, where $m$ is the sum of $m_j^T$ for all of the shapes in $\mathcal{T}$. The running time of the bottom-left and the best-fit algorithms is then $\sum_{i=1}^{n} \sum_{j=1}^{t} O((m_i + m_j^T) \log M) = O((tm + mn) \log M)$, where $M$ is the sum of $m_i$ for all of the bitmap items in $\mathcal{B}$. Considering the fact that $m$ and $M$ increase as the resolution of bitmap items becomes higher, while $t$ and $n$ are independent of the resolution, $t$ and $n$ are typically much smaller than $m$ and $M$. This implies that the running time of $O((tm + mn) \log M)$ is a significant improvement from $O(Mm \log M)$; the former is almost linear to the vertical resolution of given shapes, while the latter is quadratic to it. Similar argument is also possible to show a similar result for horizontal resolution if every given shape consists of a constant number of $x$-monotone shapes.
7. Numerical Results

The construction algorithms proposed in this paper were implemented with the C programming language and run on a Mac PC with a 1.8 GHz Intel Core i5 processor and 4 GB memory. The performance of these algorithms was tested on some types of instances that were generated by representing, in bitmap format, eight well-known benchmark instances for the irregular packing problem.

When converting these benchmark instances in which items are represented in vector format, we set the resolution of each instance to the maximum value such that the width and height (measured in the number of pixels) of the bounding box of all given items are at most 200. For each irregular item, the corresponding bitmap item in the converted instance consists of the pixels whose central point is inside of the irregular item.

We analyze the computational results in terms of both the running time and the occupation rate. For the order of the items in the bottom-left algorithm and the priority among items in the best-fit algorithm, we considered decreasing orders of width, height and area. The occupation rate for the decreasing order of height of bounding box is slightly better than the results obtained by other orderings. Consequently, we report the results obtained with the decreasing order of height of bounding box. The computational results are shown in Tables 1 to 3.

In Table 1, we show the results obtained by the bottom-left and the best-fit algorithms. The column “#pixels” reports the number of pixels required to represent all the bitmap items. The columns “Occp” and “H” show the occupation rates in % and the strip heights for the solutions obtained by the bottom-left (BL) and the best-fit (BF) algorithms. The column “Packing” reports the running time in seconds for each algorithm. When every pixel is considered as a rectangle, both the bottom-left and the best-fit algorithm return no packing results within 7200 seconds because of the large amount of data representing the bitmap items.

<table>
<thead>
<tr>
<th>instance</th>
<th>t</th>
<th>n</th>
<th>#pixels</th>
<th>Pre-Process(s)</th>
<th>Packing(s)</th>
<th>BL</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>albano</td>
<td>8</td>
<td>24</td>
<td>237096</td>
<td>480 0.53</td>
<td>15.82 0.38</td>
<td>0.11</td>
<td>76.35 798</td>
</tr>
<tr>
<td>dagli</td>
<td>10</td>
<td>30</td>
<td>109551</td>
<td>600 0.18</td>
<td>5.97 0.87</td>
<td>0.09</td>
<td>69.80 436</td>
</tr>
<tr>
<td>dighe1</td>
<td>16</td>
<td>16</td>
<td>89970</td>
<td>960 0.88</td>
<td>9.81 4.91</td>
<td>0.12</td>
<td>63.23 470</td>
</tr>
<tr>
<td>mao</td>
<td>9</td>
<td>20</td>
<td>150232</td>
<td>540 0.31</td>
<td>12.64 2.53</td>
<td>0.09</td>
<td>64.32 458</td>
</tr>
<tr>
<td>marques</td>
<td>8</td>
<td>24</td>
<td>179780</td>
<td>480 0.22</td>
<td>6.40 0.79</td>
<td>0.11</td>
<td>82.32 420</td>
</tr>
<tr>
<td>shapes</td>
<td>4</td>
<td>43</td>
<td>359370</td>
<td>240 0.14</td>
<td>28.31 0.61</td>
<td>0.16</td>
<td>57.81 1036</td>
</tr>
<tr>
<td>shirts</td>
<td>8</td>
<td>99</td>
<td>485550</td>
<td>480 0.27</td>
<td>45.40 2.29</td>
<td>0.37</td>
<td>81.58 992</td>
</tr>
<tr>
<td>swim</td>
<td>10</td>
<td>48</td>
<td>253818</td>
<td>600 0.64</td>
<td>40.79 14.26</td>
<td>0.31</td>
<td>57.17 740</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the three representation methods

Table 1 shows the computational results obtained by the two representation methods, the Scan-line representation and the Rectangle representation, and those obtained by the efficient implementation, procedure Integrate-NFP. The columns “SL-R,” “Rec-R” and “Pro-IN” show the results for the Scan-line representation, the Rectangle representation, and the procedure Integrate-NFP. Since the time complexities of the bottom-left and the best-fit algorithms are the same, we omit the running time of the best-fit algorithm. Indeed, the observed CPU times of the bottom-left and best-fit algorithms were almost the same for all tested instances. The columns in “Pre-Process” show the running times for the Rectangle representation to prepare compact representations of shapes and for procedure Integrate-NFP to prepare compact representations of NFPs for all pairs of shapes. We omit...
Algorithms for bitmap shape packing

147

the times for the Scan-line representation because they were too small to observe. For the Rectangle representation, the time limit of the SCP solver for each shape was set to 60 seconds. Observe in column “Packing” that the running time of the bottom-left algorithm that uses procedure Integrate-NFP to reduce the complexity of NFPs is much smaller than the others.

In Table 2, we show the computational results obtained by our algorithms for the case in which rotation of 180° is allowed. The columns “Occp” and “H” show the occupation rates in % and the strip heights for the solutions obtained by the bottom-left (BL) and the best-fit (BF) algorithms, and column “Time” shows the running times in seconds of the algorithms. Observe that the occupation rates obtained by the bottom-left and the best-fit algorithms for the instances with rotations of 180° allowed are always better than those without rotation with two exceptions for BF on instances “mao” and “shirts.”

Table 2: Computational results on instances with rotation of 180° allowed

<table>
<thead>
<tr>
<th>instance</th>
<th>t</th>
<th>n</th>
<th>#pixels</th>
<th>W</th>
<th>Pro-IN(s)</th>
<th>BL</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>albano</td>
<td>8</td>
<td>24</td>
<td>237096</td>
<td>343</td>
<td>1.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dagli</td>
<td>10</td>
<td>30</td>
<td>123551</td>
<td>360</td>
<td>0.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mao</td>
<td>9</td>
<td>20</td>
<td>150232</td>
<td>510</td>
<td>1.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>marques</td>
<td>8</td>
<td>24</td>
<td>179780</td>
<td>520</td>
<td>0.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shapes</td>
<td>4</td>
<td>43</td>
<td>359370</td>
<td>600</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts</td>
<td>8</td>
<td>99</td>
<td>455560</td>
<td>600</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td>10</td>
<td>48</td>
<td>253818</td>
<td>575</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithms can easily be applied to the case in which rotation is allowed [13]. In our experiments, we gave priority for orientations of every item (i.e., the rotation angles). In the bottom-left algorithm, when an item in the given order is to be packed, we try all its orientations and adopt the orientation having the leftmost BL position among the lowest ones. Similarly, in each iteration of the best-fit algorithm, we calculate the BL positions of all remaining items with respect to all orientations, and we pack the item with the orientation whose BL position has the smallest x-coordinate among those having the smallest y-coordinate.

Table 3: Computational results on large-scale instances

<table>
<thead>
<tr>
<th>instance</th>
<th>t</th>
<th>n</th>
<th>#pixels</th>
<th>W</th>
<th>Pro-IN(s)</th>
<th>BL</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts</td>
<td>8</td>
<td>99</td>
<td>485550</td>
<td>600</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts2</td>
<td>8</td>
<td>198</td>
<td>971100</td>
<td>849</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts4</td>
<td>8</td>
<td>396</td>
<td>194220</td>
<td>1200</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts8</td>
<td>8</td>
<td>792</td>
<td>388440</td>
<td>1697</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts16</td>
<td>8</td>
<td>1584</td>
<td>776880</td>
<td>2400</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shirts32</td>
<td>8</td>
<td>3168</td>
<td>1553760</td>
<td>3394</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td>10</td>
<td>48</td>
<td>253818</td>
<td>575</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim2</td>
<td>10</td>
<td>96</td>
<td>507636</td>
<td>813</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim4</td>
<td>10</td>
<td>192</td>
<td>1015272</td>
<td>1150</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim8</td>
<td>10</td>
<td>384</td>
<td>2030544</td>
<td>1626</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim16</td>
<td>10</td>
<td>768</td>
<td>4061088</td>
<td>2300</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim32</td>
<td>10</td>
<td>1536</td>
<td>8122176</td>
<td>3253</td>
<td>2.47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also generated large-scale instances by copying the shapes of the instances in Table 2. The number after the name of each instance shows the number of copies, e.g., “shirts2”
contains two copies of every item in “shirts” and hence the number $n$ of items is doubled. We report the computational results for two types of such instances in Table 3. The column “Pro-IN” shows the computation time used by procedure Integrate-NFP spent to prepare NFPs for all pairs of shapes in $T$.

Figure 11 shows an example layout obtained by the bottom-left algorithm for the instance named “shirts” with rotations of $180^\circ$ allowed. The layout in the figure is rotated $90^\circ$ clockwise, following a convention often adopted in the literature of irregular packing. The occupation rate of this layout is 82.75\%, as reported in Table 2.

Figure 11: Layout obtained for “shirts” by the bottom-left algorithm

8. Conclusions

In this paper, we proposed an efficient method for checking for overlaps and two construction algorithms for the bitmap shape packing problem. We analyzed the time complexities of our algorithms and showed that both the bottom-left and the best-fit algorithms run in $O((tM + mn) \log M)$ time when each bitmap shape can be divided into a constant number of $y$-monotone parts.

We also performed a series of experiments for instances generated from well-known benchmark instances of the irregular packing problem. The occupation rates of the packing layouts obtained by our algorithms were more than 80\% for several instances. Even for instances with more than 3000 items, represented with more than 10 million pixels in total, the bottom-left and the best-fit algorithms with our efficient implementation took less than 50 seconds. The computational results show that the proposed algorithms are especially effective for large-scale instances.

Acknowledgement

This work was partially supported by KEIT grant funded by MOTIE [Grant No. 10053204], and by JSPS KAKENHI [15K16293, 17K12981, 15H02969, 17K00038, 15K12460].

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


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

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