Journal of the Operations Research
Society of Japan
Vol. 45, No. 2, June 2002

© 2002 The Operations Research Society of Japan

A SCANLINE-BASED ALGORITHM FOR THE 2D FREE-FORM BIN PACKING PROBLEM

Hiroyuki Okano
IBM Research, Tokyo Research Laboratory

(Received April 27, 2000; Final December 4, 2001)

Abstract This paper describes a heuristic algorithm for the two-dimensional free-form bin packing (2D-FBP) problem, which is also called the irregular cutting and packing, or nesting problem. Given a set of 2D free-form bins, which in practice may be plate materials, and a set of 2D free-form items, which in practice may be plate parts to be cut out of the materials, the 2D-FBP problem is to lay out items inside one or more bins in such a way that the number of bins used is minimized, and for each bin, the yield is maximized. The proposed algorithm handles the problem as a variant of the one-dimensional bin-packing problem; i.e., items and bins are approximated as sets of scanlines, and scanlines are packed. The details of the algorithm are given, and its application to a nesting problem in a shipbuilding company is reported. The proposed algorithm consists of the basic and the group placement algorithms. The basic placement algorithm is a variant of the first-fit decreasing algorithm which is simply extended from the one-dimensional case to the two-dimensional case by a novel scanline approximation. The group placement algorithm is an extension of the basic placement algorithm with recombination of input items. A numerical study with real instances shows that the basic placement algorithm has sufficient performance for most of the instances, however, the group placement algorithm is required when items must be aligned in columns. The qualities of the resulting layouts are good enough for practical use, and the processing times required for both algorithms are much faster than those by manual nesting.

1. Introduction

In the *two-dimensional free-form bin packing* (2D-FBP) problem, which is also called the *irregular cutting and packing*, or *nesting* problem, given a set of 2D free-form items, which in practice may be plate parts, and a set of 2D free-form bins, which in practice may be plate materials from which parts are to be cut, one is asked to lay out items inside one or more bins in such a way that the number of bins used is minimized and the *yield* is maximized, where the yield for a layout in a bin is defined as the area of items over the area of the minimum rectangle parallel to the x- and y-axis containing the layout, called the *bounding rectangle.* The 2D-FBP problem is seen in a number of industries in which parts with free-form (irregular) shapes are cut from free-form or rectangular materials. For example, in the shipbuilding industry, plate parts with free-form shapes for use in the inner frameworks of ships are cut from rectangular steel plates, and in the apparel industry, parts of clothes are cut from fabric or leather.

Since the 2D-FBP problem belongs to the class of NP-hard combinatorial optimization problems, heuristic algorithms play an important role in practical applications. In the literature, heuristic algorithms for the 2D-FBP problem generally consist of procedures for approximating input items and bins, and for placing items into bins one by one and obtaining a solution. Some algorithms also include a subsequent recombination process. In the approximation of input items, representations of items are generally classified into four

types: bounding orthogonal rectangles, collections of orthogonal rectangles, simple polygons, and bitmaps (grids). A drawback of these types of representations is that, because they are two-dimensional, the subsequent placement procedure becomes complicated.

This paper proposes a new method for approximating input items and bins by *scanlines*, and for representing them by sets of intervals. A procedure for placing items is also proposed. The proposed algorithm packs sets of intervals along scanlines, instead of faces, and is shown to be efficient and practical through an intensive numerical study.

In Section 2, algorithms for the one- and two-dimensional bin-packing problems are reviewed. In Section 3, a new heuristic algorithm for the 2D-FBP problem, consisting of an algorithm for approximating input items and bins and placement algorithms, is described. Section 4 describes a numerical study using real instances obtained from a shipbuilding company. Finally, the paper is summarized in Section 5.

2. Preliminary

2.1. 1D bin-packing algorithms

Given a set P of items, a rational size $(0, 1]$ for each item, and a set of unit-capacity bins, the one-dimensional bin-packing problem is to find a partition of P into disjoint subsets such that items can be placed into the minimum number of bins; i.e., the sum of the sizes of items in each subset should be no more than 1, and the number of bins used should be minimized. This problem is known to be NP-hard [7].

One algorithm for the one-dimensional bin-packing problem is the *first-fit* algorithm. This algorithm, starting with a sequence of empty unit-capacity bins, places each item in succession into the first bin it will fit. The asymptotic worst-case performance ratio of the first-fit algorithm has been proved to be 1.7 [4]. When the input items are sorted in decreasing order of size before applying the first-fit algorithm, it is called the *first-fit decreasing* algorithm, and the bound is improved to $1.22\ldots$ [4]. The algorithms for 2D-FBP proposed in Section 3 are basically variants of the first-fit decreasing algorithm, modified for the two-dimensional case.

2.2. 2D bin-packing algorithms

Given a set P of n items and a set M of m bins whose shapes are two-dimensional, the two-dimensional bin-packing problem is to lay out items inside bins in such a way that the number of bins used is minimized and the yield is maximized. Problems of this type are obviously harder than one-dimensional bin packing, and thus NP-hard. They are also called *two-dimensional cutting stock* problems; in this case bins are called stock sheets, and items (products) are to be cut from the sheets. The shapes of items and the constraints to be considered in placing them inside bins vary according to the problem. For example, when items and bins are both rectangular, and items must be cut from bins only by orthogonal guillotine cuts, it is called the *guillotine*-cutting stock problem. For this problem, a set-covering based heuristic algorithm was proposed [12], in which a set of cutting layouts is first generated and a subset of the layouts are selected by integer programming to cover all the items to be produced.

When the shapes of the items and bins are not constrained, that is, when they may be irregular, the problem is called the two-dimensional free-form bin packing (2D-FBP) problem, or simply the nesting problem. Items in 2D-FBP are simple polygons which may be non-convex, and may contain holes inside them. Allowing interior holes is crucial in the application of 2D-FBP to the shipbuilding industry, where items (ship parts) have many holes to reduce their weight. Bins in 2D-FBP are also simple polygons which may be non-

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

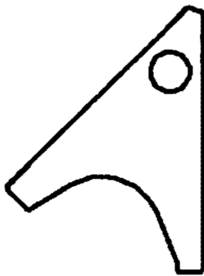


Figure 1: An input item.

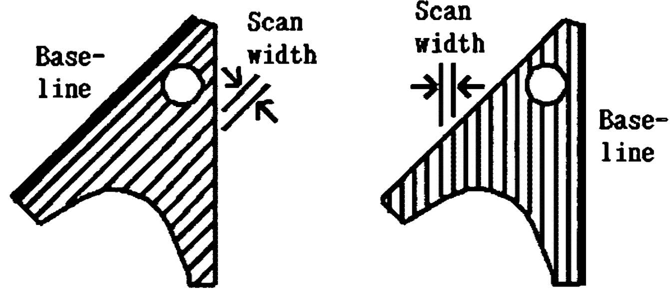


Figure 2: Scanlines in placement orientations parallel to two baselines.

convex, and may contain unusable regions inside them. For example, in the application of leather cutting, the bins (hides) are irregularly shaped, and any holes in a hide must be taken into account.

Algorithms for the 2D-FBP problem generally consist of procedures for approximating input bins and items, and for placing items into bins. One of the first attempts for 2D-FBP approximated input items as rectangles [8]. A heuristic search proposed by Albano and Sapuppo handles input items as polygons [1]. Recent studies by Daniels and Milenkovic [5, 9] and Dowsland, Dowsland and Bennell [6] also handle polygons. Qu and Sanders approximated input items as collections of orthogonal rectangles [10]. The above approaches do not allow items to contain holes. However, in practical applications, small items sometimes must be placed inside holes in large items. Some recent approaches, in which items and bins are both approximated as bitmaps (grids), satisfy this requirement. For the placement algorithms, some researchers have tried Genetic Algorithm (GA)-based approaches using large amounts of computing power. For example, a nesting system by Yamauchi and Tezuka [13] and an algorithm by Ratanapan and Dagli [11] are both GA-based. For related work, see the survey paper by Cheng, Feiring and Cheng [3].

The algorithm described in the next section approximates input items and bins by scanlines, and handles them as sets of intervals. One typical application of the algorithm is a problem involving the nesting of plate parts for shipbuilding, where the shapes of input items (ship parts) are free-form, and the shapes of input bins (material plates) are all rectangular. A convenient property of this problem is that the orientations of items to be placed into bins can be predetermined, and the number of such *placement orientations* for each item may be practically restricted to two for most input items. This is because each input item typically contains two or more long straight lines (Figure 1), and the best results are obtained when one of these lines is parallel to the x axis of a rectangular bin, where the x axis is the longer side of the bin. The proposed algorithm, taking advantage of this property, determines two placement orientations for each item, and approximates an item by scanlines along these orientations.

3. Placement Algorithms

The placement algorithms for the 2D-FBP proposed in this section have the following framework:

1. Approximate input items and bins, and represent them using sets of interval arrays.
2. Obtain an ordering of items with respect to their areas and the similarities among them.

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

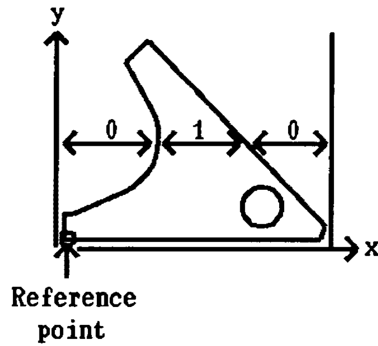


Figure 3: Intervals in run-length coding along a scanline and a reference point of an item.

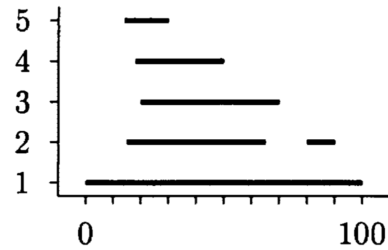


Figure 4: An example of {0,1}-intervals.

3. Select an item (or a group of items), and place it into a bin.
4. Continue Step 3 while any items remain.

3.1. Approximation and representation of items

For each input item, one or two orientations are first determined. These orientations, called the placement orientations, are used to place the item into a rectangular or irregularly shaped bin so that one of the orientations is parallel to the x axis of the bin. In determining the placement orientations of an item i, a convex hull of the item is calculated, and one or two of the longest edges in the hull are selected. The selected edges in the hull are called *baselines* and denoted as $base_i = \{1, 2\}$. Placement orientations parallel to the baselines are thus determined. Each item is then sliced along the placement orientation into strips of the same width θ that was given to the algorithm as input (Figure 2). This width refers to the size of each strip in the direction orthogonal to the baseline. The lines are called scanlines, and the width θ of strips is called the *scan width.*

An item $i \in P$, sliced parallel to one of the placement orientations, is further represented by a set of intervals in *run-length coding.* Let the number of scanlines and the width of the item i with respect to the current placement orientation be h_i and w_i, respectively. h_i is determined by the item size and the scan width θ, which is $O(1/\theta)$ assuming that the item size is a constant factor for all the items. The value of w_i is the difference between the minimum and the maximum x-coordinates of the item's edges. Arrays $a_{ij}[\,]$ $(j = 1, \ldots, h_i)$ for the run-length coding of the item i along the j-th scanline are constructed as follows: Starting from the leftmost position of the item, the length of the first portion of the scanline, which lies outside the item, is set to $a_{ij}[1]$; this portion is called a *0-interval,* and is denoted as 0 in Figure 3. The length of the next portion of the scanline, which lies inside the item, is set to $a_{ij}[2]$; this portion is called a *1-interval,* and is denoted as 1 in Figure 3, and so on. The lengths of 0- and 1-intervals are set to $a_{ij}[\,]$, one by one, ending with a 0-interval even if the length of the last 0-interval is zero. Finally, s_{ij} is set so that $2s_{ij} + 1$ is equal to the number of elements in $a_{ij}[\,]$. Arrays $a_{ij}[\,]$ are called *interval arrays.* When the $\{0, 1\}$-intervals of an item are obtained for the example in Figure 4, the number of scanlines h_i is 5, the width of item w_i is 100, and the arrays $a_{ij}[\,]$ and their sizes s_{ij} are set as follows:

$$\begin{aligned} a_{i,5}[\,] &= \{14, 16, 70\}, & s_{i,5} &= 1, \\ a_{i,4}[\,] &= \{18, 32, 50\}, & s_{i,4} &= 1, \\ a_{i,3}[\,] &= \{20, 50, 30\}, & s_{i,3} &= 1, \\ a_{i,2}[\,] &= \{15, 50, 15, 10, 10\}, & s_{i,2} &= 2, \\ a_{i,1}[\,] &= \{0, 100, 0\}, & s_{i,1} &= 1. \end{aligned}$$

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

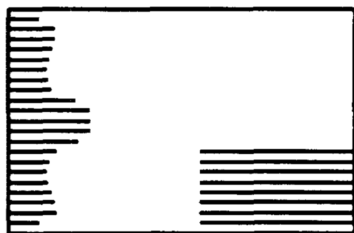


Figure 5: An example of irregularly shaped bin; a rectangular material from which some items have already been cut.

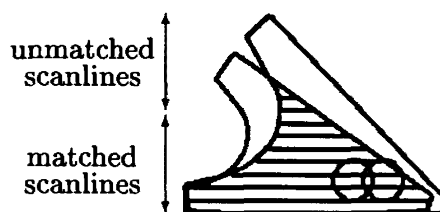


Figure 6: Matched and unmatched scanlines of two items.

Note that, for simplicity, w_i, h_i, a_{ij}, and s_{ij} are written without subscripts to specify placement orientations, although they are generated for each placement orientation, $base_i = \{1,2\}$, in actual implementations. Note also that a much smaller scan width is normally used. For example, the item in Figure 3 is sliced along 62 scanlines in the numerical study described later.

The running time complexity of this step is $O(\sum_{i\in P}(\sum_{j=1}^{h_i}(s_{ij} + \log e_i) + e_i \log e_i)) = O((n/\theta + nE)\log E)$, where e_i is the number of edges of an item i, and $E = \max_{i\in P}\{e_i\}$. The $\log e_i$ is for finding the crossing points of a scanline and the item's edges using a ray shooting data structure [2], and $e_i \log e_i$ is for constructing the convex hull and the ray shooting data structure for the item i. In the actual implementation used in the numerical study, however, searches for crossing points are performed by a naive method of $O(e_i)$-time. The required space for representing the input items, the convex hull, and the ray shooting data structure is $O(nE)$, and the space for representing the interval arrays for the input items is $O(\sum_{i\in P}\sum_{j=1}^{h_i} s_{ij}) = O(n/\theta)$.

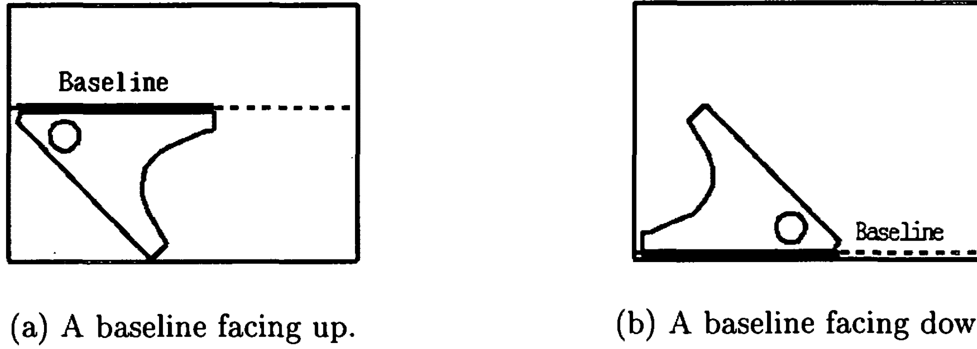
3.2. Approximation and representation of bins

The input bins may either be free-form or rectangular. When a bin has a free-form shape, a supporting line of the convex hull of the bin parallel to its diameter is selected as the x axis. When a bin is rectangular, one of the longer sides of the bin is selected as the x axis.

Each input bin $b \in M$ is sliced parallel to the x axis into strips of the scan width θ, that is, with the same distance between scanlines as that used in approximating input items. Let H_b be the number of scanlines and W_b be the width of the bin, where the width of the bin is the difference between the minimum and the maximum x-coordinates of the bin's edges. Note that both H_b and W_b are given to the algorithm as input. The bin b just after placing the i-th item is represented by interval arrays, $A_{bk}^{(i)}[\,]$ $(k = 1,\ldots,H_b)$. $S_{bk}^{(i)}$ maintains the number of 1-intervals so that $2S_{bk}^{(i)} + 1$ is equal to the number of elements in $A_{bk}^{(i)}[\,]$. The interval arrays for bins $b \in M$ are maintained in the same manner as those for items; i.e., $\sum_{j=1}^{2S_{bk}^{(i)}+1} A_{bk}^{(i)}[j] = W_b$. In the beginning, $S_{bk}^{(0)}$ is set to 0, and $A_{bk}^{(0)}[1]$ is set to W_b for $k = 1,\ldots,H_b$. When the bin has a free-form shape, interval arrays of its bounding rectangle are first generated, and 1-intervals are added to the arrays so that portions of the bounding rectangle which lie outside the bin are filled (Figure 5).

The running time complexity of this step is $O(\sum_{b\in M} H_b) = O(m/\theta)$, if all the input bins are rectangular; otherwise additional cost is required to initialize the interval arrays of the bins to represent their shapes. Note that the bin size H_b is assumed to be $O(1/\theta)$. The required space for representing the input items is $O(\sum_{b\in M} H_b S) = O(m/\theta)$, where S is the maximum number of intervals to be used for representing layouts.

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



(a) A baseline facing up. (b) A baseline facing down.

Figure 7: Two ways of placement of an item with respect to baselines.

3.3. Preprocess

An ordering π of items in P is obtained in the preprocess step with respect to their *areas* and the *similarities* among them. The area $area(i)$ of each item $i \in P$ is calculated without taking into account holes inside the item. It can be easily estimated by summing up the lengths of its scanlines; i.e., $area(i) = \sum_{j=1}^{h_i}(w_i - a_{ij}[1] - a_{ij}[2s_{ij}+1])$. The similarity $sim(i,j)$ between two items $i, j \in P$ is defined as

$$sim(i,j) = \min_{base_i \times base_j} \left\{ \frac{\#\text{ of } unmatched \text{ scanlines}}{\#\text{ of } matched \text{ scanlines}} \right\},$$

where a scanline is said to be matched if more than 80% of intervals along the scanline overlap each other without taking into account holes (Figure 6). In $sim(i,j)$, min is taken for all the combinations of baselines. The similarity has small value (< 1.0) if items are similar, 1.0 if the number of matched and unmatched scanlines are the same, and $+\infty$ if no scanline matches. Note that the above definition of similarity was chosen heuristically as it is suited for the target data set used in the numerical study described in the next section, and other definitions may be more suitable for other situations.

The ordering π is obtained in the following manner:

P1. Let $\pi := \phi$ and $k := 1$.

P2. Find the largest unselected item $i := \arg\max_{i \in P \setminus \pi}\{area(i)\}$.

P3. Find a *cluster* of similar items $P^{(k)} := \{i\} \cup \{j \in P \setminus \pi \mid sim(i,j) < 0.2\}$.

P4. Sort the items in $P^{(k)}$ in decreasing order of area, and let the resulting ordering be π_k.

P5. $\pi := \pi \cup \pi_k$. [Append π_k to π.]

P6. If $P \setminus \pi \neq \phi$ then $k := k+1$, and go to P2.

In Step P3, the similarities of the items are checked, and clusters of similar items are created. In Steps P4 and P5, the ordering is updated so that similar items are placed in consecutive positions in the ordering. Furthermore, the items in π are re-indexed in increasing numerical order so that $\pi(i) = i$. The obtained ordering π is called an *items list.*

The calculation of items' area at Step P2 requires $O(\sum_{i \in P}\sum_{j=1}^{h_i} 1) = O(n/\theta)$-time and $O(n)$-space, and finding and sorting a cluster of size $|P^{(k)}|$ costs $O(|P^{(k)}| \log |P^{(k)}|)$-time and $O(|P^{(k)}|)$-space, where $\sum_k |P^{(k)}| = n$. The total running time complexity for preprocessing for input items is, therefore, $O(n/\theta + n \log n)$, and the required space is $O(n)$.

The input bins are also sorted in decreasing order of area, and re-indexed in increasing numerical order in the list. The resulting list is called a *bins list.* The time and space required for preprocessing bins are $O(m \log m)$ and $O(m)$, respectively.

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

3.4. Basic placement algorithm

The basic placement algorithm is essentially the same as the first-fit decreasing algorithm, whose one-dimensional version was described in Section 2.1. It selects items in the items list, one by one, and examines two ways of placement for each of their baselines, $base = \{1, 2\}$; one with the baseline facing up (Figure 7(a)), and the other with the baseline facing down (Figure 7(b)). Those two patterns are denoted as $dir = \{\text{up}, \text{down}\}$. The procedure of the basic placement algorithm is as follows:

B0. Initialize sets of items $P_b := \phi$, $b = 1, \ldots, m$.

B1. Let $i := 1$. [For each item $i = 1, \ldots, n$, do the following:]

B2. Let $b := 1$. [For each bin $b = 1, \ldots, m$, do Steps B3 and B4.]

B3. Try to place the item i for all combinations of *base* and *dir*, at the *leftmost* position in the bin b, and evaluate each of the layouts.

B4. If any of placement was feasible then go to B5, otherwise let $b := b + 1$, and go to B3, where a placement is said to be feasible if the item can be placed in the bin.

B5. Select the best layout with respect to the cases of placement (*base* and *dir*), and place the item accordingly; i.e., the 1-intervals of the item i are added in the interval arrays of the selected bin b, and let $P_b := P_b \cup \{i\}$ and $b_i := b$, where b_i denotes the bin in which the item i is placed.

B6. If $i < n$ then let $i := i + 1$, and go to B2.

In Step B3, an item is placed at the leftmost position in the bin, as depicted in Figure 8 (a). Let F be a set of feasible placement positions of an item in a bin; i.e., xy-coordinates of the reference point (Figure 3) with which the item fits in the bin without overlapping existing items. The leftmost position is defined as $\arg\min_{i \in F}\{x_i\}$. When there are many such positions, the one with the smallest y-coordinate is chosen.

The placement in Step B3 is realized by three procedures: *place_left*, *containment*, and *add*. In this section, they are briefly sketched, and their details are deferred to the Appendix.

Place_left$(b, i, base, dir)$, called for each placement in Step B3, places an item i in the leftmost space in a bin b using the item's baseline specified by $base = \{1, 2\}$, facing it up or down according to $dir = \{\text{up}, \text{down}\}$. *Place_left* calls *containment* with several possible positions until it finds a position where the item will fit, and calls *add* to actually place the item at the position. *Containment*(b, i, j, k, x) (Figure 9) returns zero if 0-intervals in the bin's interval array $A_{bk}^{(i-1)}[\,]$ can contain all the 1-intervals in the item's interval array $a_{ij}[\,]$ starting at position x; otherwise it returns a promising position where the item may fit. The return value, if it is non-zero, is used for shifting the placement position of the item. In Figure 9, the specified interval arrays cannot contain each other in Step 1 because the intervals corresponding to $A[2]$ and $a[4]$ overlap each other, and the value of x_2 depicted in the figure is returned. x_2 is determined so that the interval of $a[4]$ will be placed just after the interval of $A[2]$. Note that, in this example, Step 2 with x_2 will fail again because the intervals of $a[2]$ and $A[2]$ will overlap each other, and a feasible position is found in Step 3. *Containment* is called for consecutive y-coordinates until all of a_{ij}, $j = 1, \ldots, h_i$, can be placed at the position. *Add*(b, i, j, k, x) (Figure 10) merges the specified two interval arrays $a_{ij}[\,]$ and $A_{bk}^{(i-1)}[\,]$, setting the starting point of an item's interval arrays at x.

The number of x-coordinates examined by *place_left* for the i-th item is at most the number of combinations of $\sum_{b \in M} \sum_{k=1}^{H_b} S_{bk}^{(i-1)} \le \sum_{l=1}^{i-1} \sum_{j=1}^{h_l} s_{lj}$ and $\sum_{j=1}^{h_i} s_{ij}$, i.e., $O(i/\theta^2)$. Each call to *containment* and *add* run in $O(1/\theta)$-time. Thus, the worst-case time complexity for placing n input items is $O(\sum_{i=1}^{n} i/\theta^3) = O(n^2/\theta^3)$. The actual cost, however, is smaller

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

(a) In the leftmost position by *place_left* procedure.

(b) In the bottom-most position by *place_bottom* procedure.

Figure 8: Two placement strategies. The hatched item is being placed in the leftmost and the bottom-most positions.

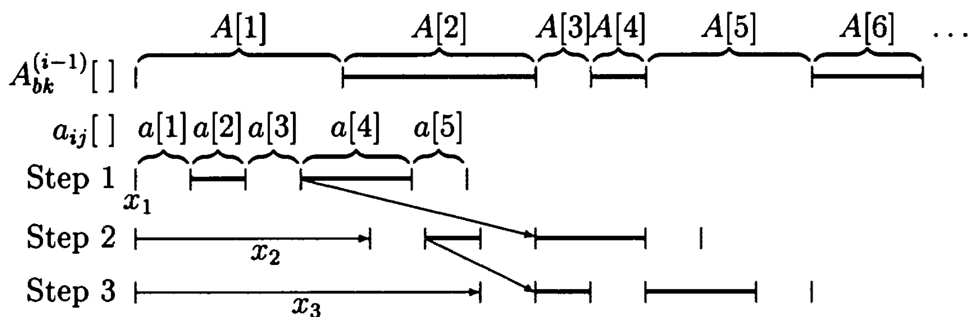


Figure 9: Shifting placement position x, in *containment*(b, i, j, k, x), to get the next promising position, and to obtain a feasible position where the item's 1-intervals will fit into the bin. $A_{bk}^{(i-1)}[\,]$ is an interval array of the bin, and $a_{ij}[\,]$ is that of the item i along a scanline.

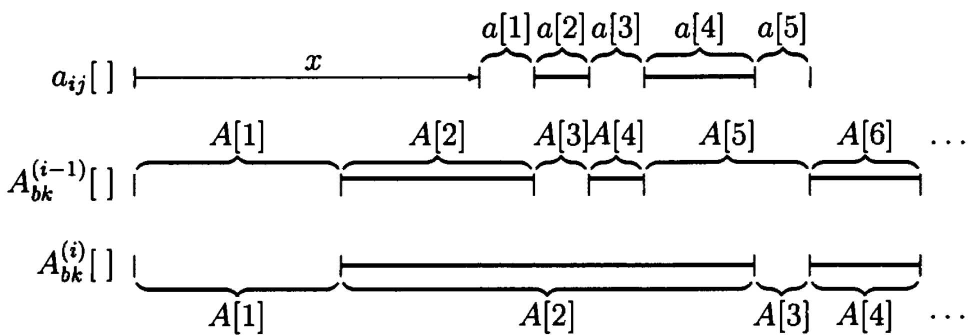
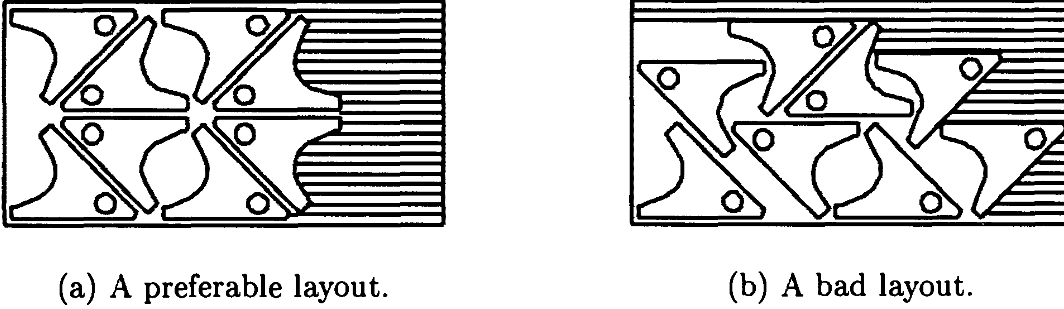


Figure 10: Merging an interval array $a_{ij}[\,]$ into an interval array $A_{bk}^{(i-1)}[\,]$, in *add*(b, i, j, k, x), to obtain $A_{bk}^{(i)}[\,]$.

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



(a) A preferable layout. (b) A bad layout.

Figure 11: To maximize the rightmost unused area.

than that because *containment* skips some x-coordinates when 1-intervals conflict (Figure 9). The required space for the placement procedure is $O(1/\theta)$.

The layouts in a bin are evaluated in Step B3, and the layout with the largest value is selected in Step B5. The evaluation function $e(b)$ to be maximized, which estimates the rightmost unused area (Figure 11) of the bin b, is defined as

$$e(b) = \sum_{k=1}^{H_b} A_{bk}[2S_{bk}^{(i)} + 1].$$

By selecting the layout with the largest value of $e(b)$ among the various placements, layouts in which a large empty space remains at the rightmost part in a bin are preferred (Figure 11(a)) over layouts in which unused space is scattered into small regions (Figure 11(b)). Note that the primary objective of the 2D-FBP problem is minimization of the number of bins used, and maximization of the yield is the secondary objective. The objective function $f(\cdot)$ to be minimized for this problem can be written as follows:

$$\begin{aligned} f(\cdot) &= nbins \times L - yield, \\ nbins &= |\{P_b \mid P_b \neq \phi, b \in M\}|, \\ yield &= \textstyle\sum_{b \in M}((\sum_{i \in P_b} area(i))/(xmax(b) \times ymax(b))), \\ xmax(b) &= \max_{k=1,\ldots,H_b}\{\textstyle\sum_{j=1}^{2S_{bk}^{(n)}} A[j]\}, \\ ymax(b) &= \arg\max_{k=1,\ldots,H_b}\{k \mid S_{bk}^{(n)} \geq 1\}, \end{aligned}$$

where L is a large value, *nbins* is the number of bins used, *yield* estimates the yield, and $xmax(b)$ and $ymax(b)$ are the x- and y-coordinates of the right and the top edges, respectively, of the bounding rectangle of the bin b. The basic placement algorithm proposed in this paper does not directly handle this objective, though it is indirectly taken into account. Note that, by the nature of *place_left* procedure, $ymax(b) \simeq H_b$ for most cases, and thus $xmax(b) \times ymax(b) \simeq W_b \times H_b - e(b)$.

3.5. Group placement algorithm

In one of the target applications of this paper in the shipbuilding industry, there are layouts created by experts that contain *aligned columns* of items, as seen in Figure 11(a) as groups of four vertically aligned items. It is therefore regarded as a good heuristic to use such patterns for some instances which contain many items with similar shapes. When the aligned columns cannot be obtained by using the basic placement algorithm introduced in the previous subsection, one can improve the solution by placing a group of a few items in the list and at the same time examining all combinations of placements for each item.

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

Groups of items are created within each cluster $P^{(k)}$ of similar items by selecting u items from the associated sorted list π_k as $\{\pi_k(1),\ldots,\pi_k(u)\}$, $\{\pi_k(u+1),\ldots,\pi_k(2u)\}$, ... Note that, in the preprocess step, the items list is generated so that clusters of similar items are adjacent in the list. The size u of the group is set to four in the numerical study described in the next section. When a group consists of two items i and $i+1$, i.e., $u=2$, for example, all combinations of placement are examined as follows:

$$\begin{array}{llll}
\{ & \mathit{place_left}(i,1,\text{up}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
\{ & \mathit{place_left}(i,1,\text{down}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
\{ & \mathit{place_left}(i,2,\text{up}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
 & & \vdots & \\
\{ & \mathit{place_left}(i,2,\text{down}), & \mathit{place_left}(i+1,2,\text{down}) & \}.
\end{array}$$

In this example for $u=2$, there are $4^u = 16$ combinations of placements.

In addition, all combinations of placements in which the first item in each cluster is placed at the *bottom-most* positions (Figure 8 (b)) are examined as follows:

$$\begin{array}{llll}
\{ & \mathit{place_bottom}(i,1,\text{up}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
\{ & \mathit{place_bottom}(i,1,\text{down}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
\{ & \mathit{place_bottom}(i,2,\text{up}), & \mathit{place_left}(i+1,1,\text{up}) & \}, \\
 & & \vdots & \\
\{ & \mathit{place_bottom}(i,2,\text{down}), & \mathit{place_left}(i+1,2,\text{down}) & \},
\end{array}$$

where the bottom-most position is defined as $\arg\min_{i\in F}\{y_i\}$. When there are many such positions, the one with the smallest x-coordinate is chosen. $\mathit{Place_bottom}(b,i,\mathit{base},\mathit{dir})$ is a procedure to place an item i in the bottom-most space in a bin b using the item's baseline specified by *base*, facing it up or down according to *dir*.

Place_bottom is defined by using the *containment* and *add* subroutines described in the last subsection, in the same manner as *place_left*. That is, it calls *containment* with several possible positions, until it finds a position where the item fits in, and calls *add* to actually place the item at the position. Its details are described in the Appendix.

The improved placement algorithm, called the group placement algorithm, is as follows:

G1. Let $k := 1$, and initialize sets of items $P_b := \phi$, $b = 1,\ldots,m$.

G2. If any group of up to u items remains in π_k, take the first group (denote it as π'), and continue placing from Step G3; otherwise, stop the process.

G3. Try to place the items in the ordering $\pi'(i)$, $i = 1,\ldots,u$, for all combinations of placements at the leftmost positions in the input bins, and evaluate each of the layouts.

G4. Try to place the items in the ordering $\pi'(i)$, $i = 1,\ldots,u$, for all combinations of placements at the bottom-most positions in the input bins for $\pi'(1)$ and at the leftmost positions for $\pi'(i)$, $i = 2,\ldots,u$, and evaluate each of the layouts.

G5. Select the best layout with respect to the placements, and place the items in π' accordingly; i.e., 1-intervals of each item $\pi'(i)$ are added in the interval arrays of the bin b into which $\pi'(i)$ was placed, and let $P_b := P_b \cup \{i\}$ and $b_{\pi'(i)} := b$.

G6. If any clusters of items remain then $k := k+1$, and go to G2; otherwise, stop the process.

Steps G3 and G4 call a subroutine based on the basic placement algorithm as follows:

B1'. Let $i := 1$. [For each item in $\pi'(i)$, $i = 1,\ldots,u$, do the following:]

B2'. Let $b := 1$. [For each bin $b = 1,\ldots,m$, do Steps B3' and B4'.]

B3'. Try to place the item $\pi'(i)$, with the specified placement and strategy, leftmost or bottom-most, into the bin b.

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

#Parts = 40 Yield = 73.4% CPU time = 11 min.

Plate #1 (3150 mm x 19480 mm)

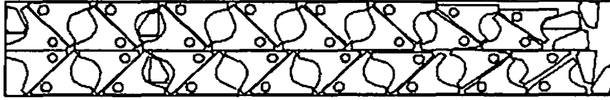


Figure 12: Layout obtained for instance A by group placement algorithm.

#Parts = 40 Yield = 72.7% CPU time = 6 min.

Plate #1 (3150 mm x 19680 mm)

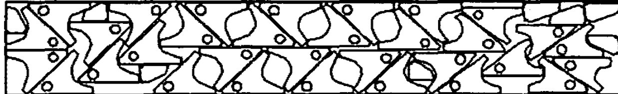


Figure 13: Layout obtained for instance A by limited group placement algorithm.

#Parts = 40 Yield = 64.7% CPU time = 1 min.

Plate #1 (3150 mm x 19960 mm)

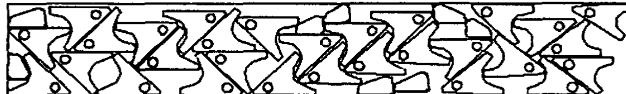


Plate #2 (2450 mm x 2530 mm)

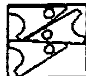


Figure 14: Layout obtained for instance A by basic placement algorithm.

#Parts = 76 Yield = 80.4% CPU time = 1 min.

Plate #1 (2775 mm x 17140 mm)

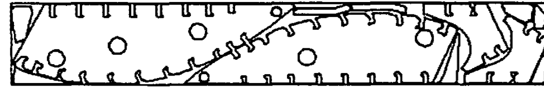


Plate #2 (2775 mm x 16930 mm)

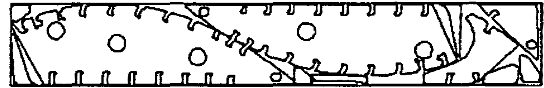


Plate #3 (3000 mm x 15120 mm)

Figure 15: Layout obtained for instance B by basic placement algorithm.

#Parts = 56 Yield = 75.8% CPU time = 1 min.

Plate #1 (3925 mm x 16620 mm)

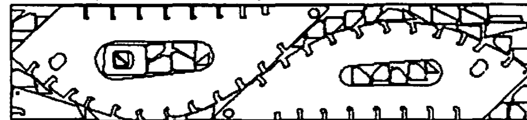


Plate #2 (3700 mm x 18430 mm)

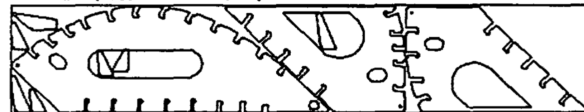


Plate #3 (2625 mm x 13460 mm)

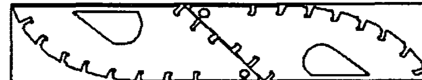


Figure 16: Layout obtained for instance C by basic placement algorithm.

B4'. If the placement was feasible, tentatively place the item $\pi'(i)$ into the bin b; otherwise let $b := b + 1$, and go to B3'.

B5'. If $i < u$ then let $i := i + 1$, and go to B2'.

B6'. Evaluate the layout, undo all the placements, and return.

Note that two layouts are examined in Steps G3 and G4 for each combination of placement, one by calling *place_left* for all the items in the group, and the other by calling *place_bottom* for the first item. The effects of the group placement algorithm and the use of the bottom-most strategy will be considered in the next section.

The group placement algorithm places groups of items one by one and terminates when all the groups have been processed; i.e., it is a deterministic greedy heuristic. The numerical study in the next section will show that solutions obtained by the algorithm are sufficiently practical for use in a shipbuilding company. If the solutions are not good enough, however, a local search or a meta-heuristic-based recombination can be applied to them.

The time and space complexities of the proposed group placement algorithm are the same as those of the basic placement algorithm, because *place_left* and *place_bottom* have the same time and space complexities, and the number of combinations in each group of

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

items, 4^u, can be regarded as a constant.

The proposed two algorithms, the basic and the group placement algorithms, can be naturally extended to the three-dimensional case, in which items and bins can be sliced into layers and each layer can be approximated by scanlines.

4. Numerical study

A numerical study was carried out by using real instances obtained from a shipbuilding company. In the company, plate parts for building the inner frameworks of ships are grouped by thickness and specification, and a nesting problem for each group is solved manually. Solving the nesting problem involves finding appropriate sizes of material plates (bins) among given standard sizes. In this numerical study, it was assumed that the groups of ship parts (items) and the appropriate sizes of the material plates (bins) are given for each instance. CPU times were measured on an IBM RS/6000 with a 332-MHz CPU.

Figure 12 shows the layout for instance A obtained by the group placement algorithm with the group size u set to four. Figures 13 and 14 show the layouts for the same instance obtained by the group placement algorithm without using the bottom-most strategy (Step G4 of the group placement algorithm) and by the basic placement algorithm, respectively. The yield and the CPU time are shown in each figure. The plate sizes shown in the figures are of the bounding rectangles of the layouts. Figure 12 shows the aligned columns of parts generated by the group placement algorithm, and Figure 13 shows that the bottom-most strategy is necessary to obtain aligned columns for this instance. Figures 15 and 16 show the layouts for instances B and C obtained by the basic placement algorithm. These figures show that the basic placement algorithm has sufficient performance when the aligned columns as seen in Figure 12 are not needed.

4.1. Comparison with another method

The above experiments show that when an aligned structure of items is required for obtaining a good layout, a special search mechanism is needed in addition to a simple first-fit decreasing type algorithm; i.e., the basic placement algorithm. Dowsland, Dowsland and Bennell [6] proposed a local improvement method, called the *jostling* approach, for a type of 2D-FBP problem. In their problem, the number of placement orientations for each item is set to one, and not even a 180° rotation of items (Figure 7) is considered. Although their approach takes advantage of the restriction of rotation of items, in this paper, their approach is compared to the group placement algorithm using the problem setting of this paper; i.e., up to two placement orientations and 180° rotations are considered for each placement orientation.

The jostling approach, as does the group placement algorithm, assumes an existing simple placement algorithm which places items at the leftmost positions in each bin. Starting with a layout, the items are ordered in increasing or decreasing order of the x-coordinates of their leftmost or rightmost points, respectively, and a layout is created using the leftmost or the rightmost placement strategy, respectively. This process is continued for a fixed number of iterations.

The jostling approach can be emulated by using the basic placement algorithm, which always uses the leftmost placement strategy, as follows:

J1. Create an ordering π of the input items (with respect to their areas and similarities between them).

J2. Call the basic placement algorithm with the ordering π, and obtain a layout.

J3. Let x_i be x-coordinates of the rightmost points of items in the input bins. Re-order the items in decreasing order of $x_i + \sum_{b=1}^{b_i-1} W_b$. Let the resulting ordering be π.

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

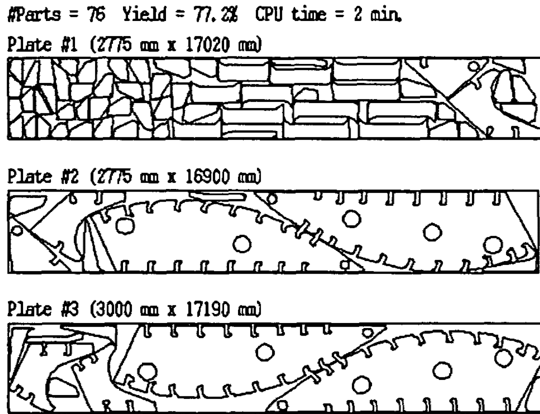


Figure 17: Layout obtained for instance B after the second iteration of the jostling procedure.

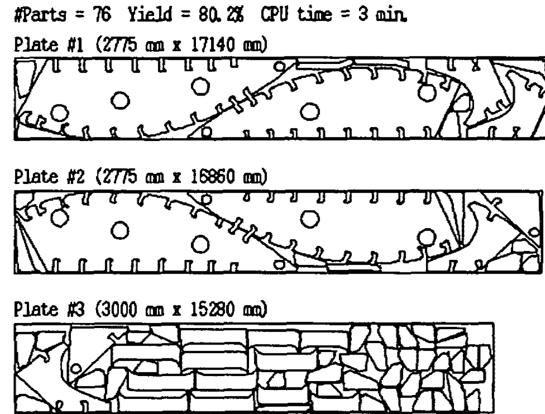


Figure 18: Layout obtained for instance B after the third iteration of the jostling procedure.

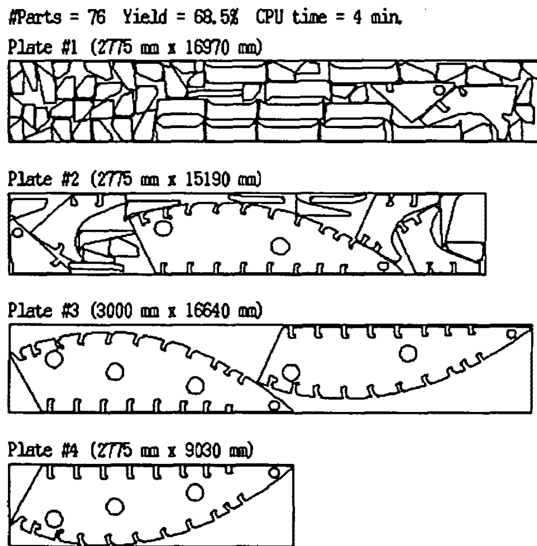


Figure 19: Layout obtained for instance B after the fourth iteration of the jostling procedure.

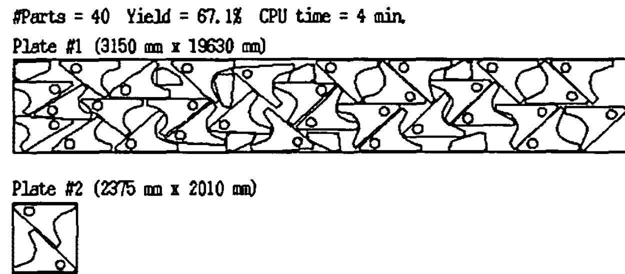


Figure 20: Best layout obtained for instance A by the jostling procedure.

J4. If the terminal condition is met then stop the process; otherwise go to J2.
In the experiments, a total of ten calls to the basic placement algorithm were made for each instance. The instances A and B were used for the comparison. The instance C was not used because the sizes of its input bins differ so much, and the above procedure cannot emulate the jostling approach very well.

Figures 17 through 19 show results of the second, third, and fourth iterations of the jostling procedure for the instance B. Large items are packed first in the odd iterations, and small items are packed first in the even iterations. The subsequent iterations could not improve the layout obtained at the first iteration (Figure 15), and the qualities of solutions tended to become worse as the iterations proceeded.

Figure 20 shows the best result by the jostling procedure for the instance A, which was obtained at the sixth iteration. The number of bins used was two for all the results. The aligned columns obtained by the group placement algorithm (Figure 12) were not created

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

Table 1: Time and space complexities of the algorithm.

Algorithm steps	Time complexity	Space complexity
Representation of items	$O((n/\theta + nE)\log E)$	$O(nE + n/\theta)$
Representation of bins	$O(m/\theta)$	$O(m/\theta)$
Preprocess step for items	$O(n/\theta + n\log n)$	$O(n)$
Preprocess step for bins	$O(m\log m)$	$O(m)$
Placement algorithms	$O(n^2/\theta^3)$	$O(1/\theta)$
Total	$O(n^2/\theta^3 + (n/\theta + nE)\log E)$	$O(n/\theta + nE)$

n: number of input items, m: number of input bins ($\ll n$), θ: a scan width, E: maximum number of edges of items.

by the jostling procedure. These results show that the jostling procedure does not converge to good solutions under the experimental setup of this paper. It seems rotation of items is crucial for the jostling approach.

4.2. Discussion

Using the basic and the group placement algorithms, solutions of sufficiently high quality for practical use were obtained for the tested instances when appropriate sizes of bins were specified. The obtained layouts have qualities comparable with those of layouts created by human experts, and the required CPU times are much faster than those required for manual nesting.

The comparison of the proposed algorithms and one of existing algorithms called the jostling approach showed that the basic placement approach may be improved by local search, however, a type of local search which does not consider groups of similar items cannot find good solutions for special instances in which aligned columns of items are required. The group placement algorithm is one of the suitable options for such a case.

As described in Section 3, the placement algorithms proposed in this paper have the worst-case time complexity of $O(n^2/\theta^3)$, where n is the number of items, and θ is the input scan width. In practice, however, by using promising x-coordinates returned by the *containment* subroutine, the algorithms run faster than the worst-case time, and it is estimated as $O(n^2/\theta^2)$. In all the experiments described above, the scan width θ was set to 25 *mm*. To investigate the effect of the change in θ to the running time, θ was changed to 10 *mm*, and the group placement algorithm was applied to the instance A. Then the running time was increased from 11 minutes to 59 minutes, which indicates that the effect of θ is less than quadratic in the computational cost.

When the scan width was set to 250 *mm*, the resulting layouts include overlapping items when the layouts were evaluated using the original edges of items, which means, the approximation of the items was not precise enough. The appropriate scan width should be determined so that all of the input items and bins are approximated with enough precision, and the running time of the program is reasonable. In the experiments in the previous subsection, the scan width was chosen heuristically.

5. Conclusion

A new heuristic algorithm for the two-dimensional free-form bin packing (2D-FBP) problem was described. The algorithm approximates input items and bins by scanlines, and handles the 2D-FBP problem as a variant of the one-dimensional bin-packing problem.

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

The algorithm consists of preparation, a basic placement algorithm, and a group placement algorithm, whose time and space complexities are shown in Table 1. The basic placement algorithm is a two-dimensional extension of the first-fit decreasing algorithm, which is known to be efficient for the one-dimensional case. The group placement algorithm is an improved version of the basic placement algorithm that examines recombination of a few similar items.

A numerical study was carried out, using real instances obtained from a shipbuilding company, and it was shown that the proposed algorithm can find layouts of ship parts comparable to those obtained by human experts, and the CPU times required by the algorithm are much shorter than the times required for manual nesting.

For future study, it is planned to improve the algorithm by slightly rotating the items in the placement procedures. It is noted that the approach used in the proposed algorithm can also be applied to the three-dimensional bin-packing problem, which appears, for example, in data preparation for 3D rapid-prototyping machines.

Acknowledgment

The author would like to thank the anonymous editors and referees for their valuable comments on the original version of this paper.

Appendix

Figures 21 through 24 are pseudocode for the procedures used in the described basic and group placement algorithms. The parameter b refers to a bin, i refers to an item to be placed into the bin, and the parameters *base* and *dir* refer to a placement. *Base* describes a baseline (1 or 2 if the item has two baselines), and $dir = \{\text{up}, \text{down}\}$ describes that the baseline should be facing up or down (Figure 7). In the pseudocode, for simplicity, *base* and *dir* have been omitted, and a placement with the baseline facing down is assumed.

The notations used in the pseudocode are:

w_i	Width of an item i,
h_i	Number of scanlines in the item i for the selected baseline,
$a_{ij}[2s_{ij}+1]$	Interval arrays of the item i where $j = 1, \ldots, h_i$,
s_{ij}	Number of 1-intervals in $a_{ij}[\,]$,
W_b	Width of a bin b,
H_b	Number of scanlines in the bin b,
$A_{bk}^{(i)}[2S_{bk}^{(i)}+1]$	Interval arrays of the bin b just after placing the i-th item in the items list, where $k = 1, \ldots, H_b$,
$S_{bk}^{(i)}$	Number of 1-intervals in $A_{bk}^{(i)}[\,]$.

Place_left (Figure 21) and *Place_bottom* (Figure 22) find the leftmost and the bottommost space, respectively, in the bin b into which the given item i will fit, and adds the 1-intervals of the item to the bin's interval arrays (Figure 8). They call the *check* and *place* subroutines to find empty spaces into which the items may fit, and add 1-intervals in the arrays.

The *check* and *place* procedures call subroutines for checking the containment of interval arrays and for merging two interval arrays, where the interval arrays are able to contain each other if no 1-intervals from the two arrays overlap each other. $Check(b, i, x, y)$ (Figure 23) calls the $containment(b, i, j, k, x)$ procedure (Figure 9), for each scanline $j = 1, \ldots, h_i$ of the item i and $k = y + j - 1$ of the bin b, to check if the interval arrays along the scanlines can contain each other. $Place(b, i, x, y)$ (Figure 24) calls the *add* procedure for each scanline of the item i and the bin b just as the *check* procedure does, where $add(b, i, j, k, x)$ (Figure 10) merges $a_{ij}[\,]$ into $A_{bk}^{(i-1)}[\,]$ starting at position x.

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

```
 1: function place_left(b, i, base, dir) : boolean;  (* Place an item i in a bin b *)
 2:  var j, x, y, pos: integer;                       (*  with the specified placement. *)
 3:      next: array [1..H_b - h_i + 1] of integer;   (* Array of x positions in the bin. *)
 4:  begin
 5:  for j := 1 to H_b - h_i + 1 do next[j] := 1;     (* Initialize next[ ]. *)
 6:  while true do                                    (* Main loop. *)
 7:   begin
 8:   x := W_b;                                       (* Set x a large value (width of the bin). *)
 9:   for j := 1 to H_b - h_i + 1 do                  (* Find the smallest value in next[ ], *)
10:    if x > next[j] then                            (*  which is the leftmost position *)
11:     begin x := next[j]; y := j; end;              (*  of the empty area. *)
12:   if x > W_b - w_i + 1 then                       (* If there is no space where the item will fit, *)
13:    return false;                                  (*  return false. *)
14:   pos := check(b, i, x, y);                       (* Check if the item fits at position (x, y). *)
15:   if pos = 0 then                                 (* If the item fits at (x, y), *)
16:    begin place(b, i, x, y);                       (*  place it there, *)
17:          return true; end;                        (*  and return true. *)
18:   next[y] := pos;                                 (* Save the position. *)
19:   end;
20:  end;
```

Figure 21: Pseudocode for placing an item at the leftmost position in a bin.

```
 1: function place_bottom(b, i, base, dir) : boolean; (* Place an item i in a bin b *)
 2:  var x := 1, y, pos: integer;                     (*  with the specified placement. *)
 3:  begin
 4:  for y := 1 to H_b - h_i + 1  do                  (* Scan the bin from the bottom to the top. *)
 5:   begin
 6:   while x < W_b - w_i + 1  do                     (* Scan the bin from the left to the right. *)
 7:    begin
 8:    pos := check(b, i, x, y);                      (* Check if the item fits at position (x, y). *)
 9:    if pos = 0 then                                (* If the item fits at (x, y), *)
10:     begin place(b, i, x, y);                      (*  place it there, *)
11:           return true; end;                       (*  and return true. *)
12:    x := pos;                                      (* The next x position. *)
13:    end;
14:   end;
15:  return false;                                    (* There is no space where the item will fit. *)
16:  end;
```

Figure 22: Pseudocode for placing an item at the bottom-most position in a bin.

```
 1: function check(b, i, x, y : integer) : integer;  (* Check if an item i fits at (x, y) in a bin b. *)
 2:  var j, pos: integer;
 3:  begin
 4:  for j := 1 to h_i do                             (* Scan the item from the bottom to the top. *)
 5:   begin
 6:    pos := containment(b, i, j, y + j - 1, x);     (* Check for intervals along j-th scanline. *)
 7:    if pos > 0 then                                (* If the item does not fit in the bin, return a *)
 8:     return pos;                                   (*  promising x position to try later by *)
 9:   end;                                            (*  the caller. *)
10:  return 0;                                        (* If the item fits at (x, y), return zero. *)
11:  end;
```

Figure 23: Pseudocode for checking if the given item fits in the given bin.

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

```
1: procedure place(b, i, x, y : integer);      (* Set an item i at (x,y) in a bin b. *)
2:   var j: integer;
3:   begin
4:   for j := 1 to h_i do                      (* Scan the item from the bottom to the top. *)
5:     add(b, i, j, y + j - 1, x);             (* Add 1-intervals of the item along j-th *)
6:   end;                                      (*  scanline to the (y + j - 1)-th interval *)
                                               (*  array of the bin b. *)
```

Figure 24: Pseudocode for checking if the given item fits in the given bin.

References

[1] A. Albano and G. Sapuppo: Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-10-5** (1980) 242-248.

[2] B. Chazelle and L. J. Guibas: Visibility and intersection problems in plane geometry. *Discrete Computational Geometry*, **4** (1989) 551-581.

[3] C. H. Cheng, B. R. Feiring and T. C. E. Cheng: The cutting stock problem – a survey. *International Journal of Production Economics*, **36** (1994) 291-305.

[4] E. G. Coffman, Jr., M. R. Garey and D. S. Johnson: Approximation algorithms for bin packing: a survey. In D.S. Hochbaum (ed.): *Approximation algorithms for NP-hard problems* (PWS Publishing Company, 1997), 46-93.

[5] K. Daniels and V. J. Milenkovic: Column-based strip packing using ordered and compliant containment. *Proc. 1st ACM Workshop on Applied Computational Geometry*, (1996) 33-38.

[6] K. A. Dowsland, W. B. Dowsland and J. A. Bennell: Jostling for position: local improvement for irregular cutting patterns. *Journal of the Operational Research Society*, **49** (1998) 647-658.

[7] M. R. Garey and D. S. Johnson: *Computers and Intractability - A Guide to the Theory of NP-Completeness.* (W. H. Freeman and Company, New York, NY, 1979).

[8] M. J. Haims: On the optimum two-dimensional allocation problem. Ph.D. Dissertation, Department of Electrical Engineering, New York University, 1966.

[9] V. J. Milenkovic: Rotational polygon containment and minimum enclosure. *Proc. 14th ACM Symposium on Computational Geometry*, (1998) 1-8.

[10] W. Qu and J. L. Sanders: A nesting algorithm for irregular parts and factors affecting trim losses. *International Journal of Production Research*, **25** (3) (1987) 381-397.

[11] K. Ratanapan and C. H. Dagli: An object-based evolutionary algorithm: the nesting solution. *Proc. 1998 IEEE Conference on Evolutionary Computation*, (1998) 581-586.

[12] P. Y. Wang: Two algorithms for constrained two-dimensional cutting stock problems. *Operations Research*, **31** (3) (1983) 573-586.

[13] S. Yamauchi and K. Tezuka: Automatic nesting system by use of genetic algorithm. *Journal of the Society of Naval Architects of Japan*, **178** (5-21) (1995) 707-712. (in Japanese)

Hiroyuki Okano
IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato
Kanagawa-ken 242-8502, JAPAN
E-mail: okanoh@jp.ibm.com

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