January 18, 1999
RT0286
Computer Science 14 pages

Research Report

An Approximation Algorithm for the 2D Free-Form Bin Packing Problem

H. Okano

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Abstract

This paper proposes an efficient and practical approximation algorithm for the two-dimensional free-form bin packing (2D-FBP) problem, which is also called the free-form cutting stock, cutting and packing, or nesting problem. In the 2D-FBP 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, you are asked to lay out items inside one or more bins in such a way that the number of bins used is minimized. The proposed algorithm handles the problem as a variant of the one-dimensional bin-packing problem; that is, 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.

Keywords

bin-packing problem, cutting stock problem, cutting and packing problem, nesting problem

1. Introduction

In the *two-dimensional free-form bin packing* (2D-FBP) problem, which is also called the free-form *cutting stock*, *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, you are 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 (area of items over area of bounding rectangles of layouts) is maximized. 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 materials.

Since the 2D-FBP problem belongs to the class of NP-hard combinatorial optimization problems, which means that there is no hope of finding polynomial-time exact algorithms unless $P = NP$, approximation algorithms play an important role in practical applications. In the literature, approximation 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 and grids, respectively. A procedure for placing items is also proposed. The proposed algorithm packs 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 approximation algorithm for the 2D-FBP problem, consisting of an algorithm for approximating input items and bins and a placement algorithm, is proposed. 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

In the one-dimensional bin-packing problem, given a set of items, a rational size $[0, 1]$ for each item, and a set of unit-capacity bins, you are asked to find a partition of the set of items into disjoint subsets such that items can be placed in the minimum number of bins; that is, 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 [1].

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 [2]. 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 $1.22\ldots$ [2]. The algorithm for 2D-FBP proposed in Section 3 is basically a variant of the first-fit decreasing algorithm, modified for the two-dimensional case.

2.2. 2D bin-packing algorithms

In two-dimensional bin-packing problems, given a set of items and a set of bins whose shapes are two-dimensional, you are asked to lay out items inside bins in such a way that the number of used bins is minimized and the yield (area of items over area of bounding rectangles of layouts) 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. This problem, for example, can be formulated as set covering 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 [3].

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. Algorithms for the 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 [4]. A heuristic search proposed by Albano and Sapuppo handles input items as polygons [5]. Recent studies by Daniels and Milenkovic [6,7] also handles polygons. Qu and Sanders approximated input items as collections of orthogonal rectangles [8]. The above approaches do not allow items to contain holes; however, in practical applications, small items are sometimes required 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 vast amounts of computing power. For example, a nesting system by Yamauchi and Tezuka [9] and an algorithm by Ratanapan and Dagli [10] are both GA-based. For related work, see a survey paper by Cheng, C., Feiring, and Cheng, T. [11].

The algorithm described in the next section approximates input items and bins by scanlines, and handles items as sets of intervals. One typical application of the algorithm is a problem involving nesting of plate parts for shipbuilding, where the shapes of input items (parts) are free-form, and the shapes of input bins (material plates) are all rectangular. A good property of this problem is that the directions of items to be placed in bins can be predetermined, and the number of such *placement directions* for each item may be practically restricted to two. This is because each input item typically contains two or more long straight lines (Fig. 1), and a good result is 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 directions for each item, and approximates an item by scanlines in these directions.

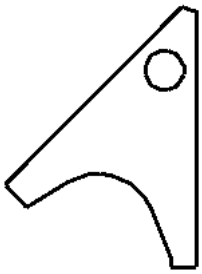
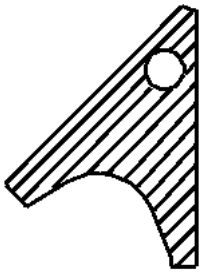


Fig. 1. An input item



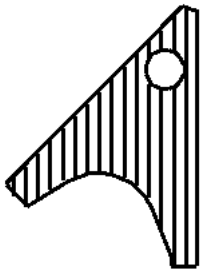


Fig. 2. Scanlines in two placement directions

3. Approximation and Placement Algorithms

3.1. Approximation and representation of items

For each input item P_i, one or two directions are first determined. These directions, called the placement directions, are used to place the item in a rectangular bin so that one of the directions is parallel to the x axis of the bin. In determining the placement directions, a convex hull of each item is calculated, and one or two of the longest edges in the hull are selected. Lines that contain the selected edges are called *baselines*, and placement directions parallel to the baselines are determined. Each item is then sliced along the placement direction into strips of the same width (Fig. 2). The sliced lines are called scanlines, and the width of strips is called the *scan width*.

An item P_i, sliced parallel to one of the placement directions, is further represented by a *run-length code*. Let the number of scanlines be r_i, and let the width of P_i in the current placement direction be l_i. Arrays $R_{ij}[\,]$ $(j = 1, 2, \ldots, r_i)$ for run-length coding of P_i along the j-th scanline are constructed as follows: Starting from the leftmost position of P_i, the length of the first portion of the scanline, which lies outside P_i, is set to $R_{ij}[1]$; this portion is called a *0-interval*, and is denoted as 0 in Fig. 3. The length of the next portion of the scanline, which lies inside P_i, is set to $R_{ij}[2]$; this portion is called a *1-interval*, and is denoted as 1 in Fig. 3, and so on. The lengths of 0- and 1-intervals are set to $R_{ij}[\,]$, 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 $s_{ij} \times 2 + 1$ is equal to the number of elements in R_{ij}. Arrays R_{ij} are called *run-length arrays*. When P_i's $\{0, 1\}$-intervals are obtained, for example, as in Fig. 4, the number of scanlines r_i is 4, the width of item l_i is 100, and arrays $R_{ij}[\,]$ and their sizes s_{ij} are set as follows:

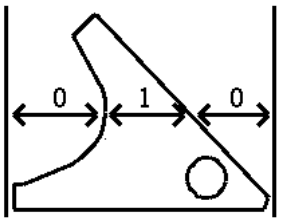


Fig. 3. Intervals in run-length coding along a scanline

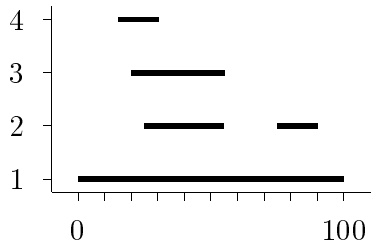


Fig. 4. An example of {0,1}-intervals

$$
\begin{aligned}
R_{i,4}[\,] &= \{15, 15, 70\}, & s_{i,4} &= 1,\\
R_{i,3}[\,] &= \{20, 35, 45\}, & s_{i,3} &= 1,\\
R_{i,2}[\,] &= \{25, 30, 20, 15, 10\}, & s_{i,2} &= 2,\\
R_{i,1}[\,] &= \{0, 100, 0\}, & s_{i,1} &= 1.
\end{aligned}
$$

Note that, for simplicity, l_i, r_i, R_{ij}, and s_{ij} are written without subscripts to specify placement directions, although they are generated for each placement direction (baseline) in actual implementations.

3.2. Approximation and representation of bins

Input bins may either be free-form (irregular) or rectangular. When a bin is irregularly shaped, a direction in which the width is greatest is selected as the x axis of the bin. When a bin is rectangular, one of the longer sides of the bin is selected as the x axis.

Each input bin M is sliced parallel to the x axis into strips of the scan width, that is, with the same distance between scanlines as that used in approximating input items. (A bin is denoted without subscript, for simplicity, although there may be more than one input bin.) Let the number of scanlines be b, and the width of the bin be a. Then M is represented as b *bitmap arrays* of size a:

$$
\boxed{
\begin{array}{c}
B_1[1, 2, \ldots\ldots, a],\\
B_2[1, 2, \ldots\ldots, a],\\
\ldots\\
B_b[1, 2, \ldots\ldots, a].
\end{array}
}
$$

Each element of the bitmap arrays has a value of zero (0) if the position is not occupied by an item, and a value of 1 if the position is occupied by an item. When the bin has a free-form

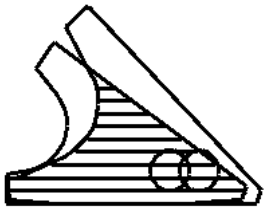


Fig. 5. Similarity between two items

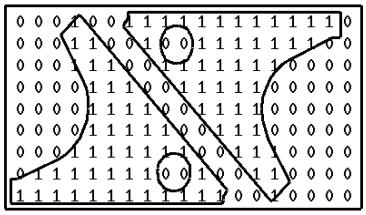


Fig. 6. Placing items in a bitmap

shape, bitmap arrays of its bounding rectangle are first generated, and positions in the array that lie outside of the bin are set to 1.

3.3. Preprocess

Input items P_i are sorted in decreasing order of area. The area of each item can be easily estimated by summing up the lengths of its scanlines; that is, the lengths of 1-intervals in $R_{ij}[\,]$ for $j = 1, 2, \ldots, r_i$. Furthermore, the similarities of items are checked, and the list is modified so that similar items are placed in consecutive positions in the list. The similarity between two items can be easily checked by summing up the lengths of 1-intervals that overlap each other (Fig. 5). Items are re-indexed in increasing numerical order in the list, and the resulting list is called an *items list*.

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*.

3.4. Basic placement algorithm

The *basic placement algorithm* is as follows:

1. If any items remain in the sorted items list, take the first item, and continue placing from Step 2; otherwise, stop the process.
2. Select the bitmap of the first bin in the input bins list.
3. Try to place the item, with various alternative placements, at the bottom left position of the bitmap, and evaluate the layout.
4. If the item cannot be placed in the bitmap because there is insufficient space, select that of the next bin in the list, and go to Step 3.
5. Select the most preferable layout with respect to the cases of placement, and place the item accordingly.
6. Go to Step 1.

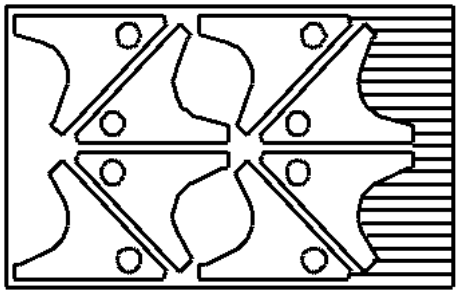


Fig. 7. To minimize the rightmost unused area

In Step 3, for each placement direction of the item, two placements are examined, one with the baseline facing up, and the other with the baseline facing down. If reversal of items is allowed, two corresponding extra cases may be added. After placement of an item, 1's are placed in a bitmap as shown in Fig. 6.

Figures 13 and 14 in Appendix show the procedures, $place\_left$ and $place\_bottom$, for placing an item P_i into a bin M for one case of placement direction in Step 3. They correspond to two strategies of the basic placement algorithm: *leftmost* and *bottom-most*. Thanks to the run-length coding of items, both procedures can efficiently find empty space in a current bin into which a given item may fit.

$Place\_left$ is called for each placement direction in Step 3, and the resulting layouts are evaluated in Step 5. $Place\_bottom$ is called in the *group placement algorithm* introduced in the next subsection. In Step 5, the most *preferable* layout among those obtained for all placements is selected. In this step, an objective function is calculated for each layout, and the layout with the smallest value is selected. The objective function, which estimates the *waste area* of a layout, is defined to minimize the rightmost unused area (Fig. 7).

The basic placement algorithm is essentially the same as the first-fit decreasing algorithm, whose one-dimensional version was described in Section 2.1. The worst-case time complexity of a call for $place\_left$ or $place\_bottom$ is $O(ab)$, which is the bitmap (grid) size of the current bin. The actual time complexity, however, is smaller than that if the current bin contains empty areas.

3.5. Group placement algorithm

When aligned columns of items as seen in Fig. 8 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 at the same time examining all combinations of

placements for each item. The group of items are selected from top of the sorted items list in such a way that items in the group are similar. Note that, in the preprocess, the items list is generated that groups of similar items are consecutive in the list.

When a group consists of items p_i and p_{i+1}, for example, all combinations of placements are examined as follows:

$$
\begin{gathered}
place\_left\ (i,1,1),\ place\_left\ (i+1,1,1),\\
place\_left\ (i,1,2),\ place\_left\ (i+1,1,1),\\
place\_left\ (i,2,1),\ place\_left\ (i+1,1,1),\\
\vdots\\
place\_left\ (i,2,2),\ place\_left\ (i+1,2,2).
\end{gathered}
$$

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

1. If any groups of items remain in the sorted items list, take the first group, and continue placing from Step 2; otherwise, stop the process.

2. Try to place items in the group, with all the combinations of placements, by using the basic placement algorithm with the leftmost strategy. (Replace the *sorted items list* in the description of the basic placement algorithm in the previous subsection with the *group of items*.)

3. Try to place items in the group in the same way as in Step 2. This time change the strategy of the first item to bottom-most.

4. Select the most preferable layout with respect to the combination of placements.

5. Go to Step 1.

Note that two layouts are examined for each combination of placements; one by calling $place\_left$ for all items in the group (Step 2), and the other by calling $place\_bottom$ for the first item and calling $place\_left$ for subsequent items (Step 3). Effects of the group placement algorithm and the use of the bottom-most strategy will be shown in Section 4.

The group placement algorithm places groups of items one by one and terminates when all the groups have been processed; that is, it is a deterministic greedy heuristic. A numerical study in the next section will show that solutions obtained by the algorithm are sufficiently practical for instances 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 worst-case time complexity of the proposed placement algorithm is the same as that of the basic placement algorithm, because the number of combinations in each group of items is constant.

The proposed two algorithms, the basic placement 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 shipbuilding company. In the company, plate parts for building 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 groups of plates (items) and appropriate sizes of material plates (bins) are given for each instance. CPU times were measured on an IBM RS/6000 model 7015-R30 with a PowerPC 601 112-MHz CPU.

Figure 8 shows the layout for instance A obtained by the group placement algorithm in which group sizes are up to four. Figures 9 and 10 show the layouts for the same instance obtained by the group placement algorithm without using the bottom-most strategy (Step 3 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 layouts. Figure 8 shows aligned columns of parts generated by the group placement algorithm, and Fig. 9 shows that the bottom-most strategy is necessary to obtain aligned columns for this instance. Figures 11 and 12 show the layouts for instances B and C obtained by the basic placement algorithm. These figures show that the basic placement algorithm has a sufficient performance when aligned columns as seen in Fig. 8 are not needed.

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. Because the implementations of other algorithms are not known to the author, the results are not compared; however, the proposed algorithm seems fast enough. The reason for this that it is a deterministic greedy heuristic, whereas meta-heuristic-based algorithms, such as GA-based nesting systems [9,10], require large numbers of iterations.

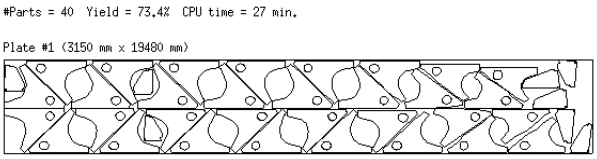


Fig. 8. Layout obtained for instance A by group placement algorithm

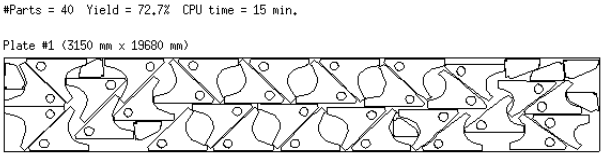


Fig. 9. Layout obtained for instance A by limited group placement algorithm

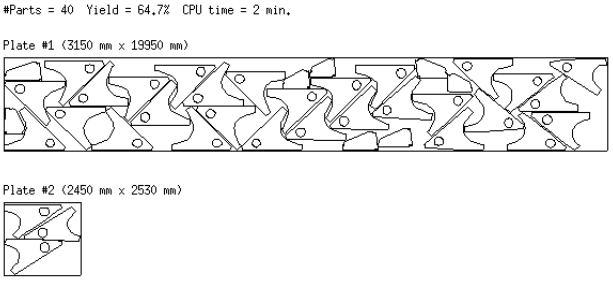


Fig. 10. Layout obtained for instance A by basic placement algorithm

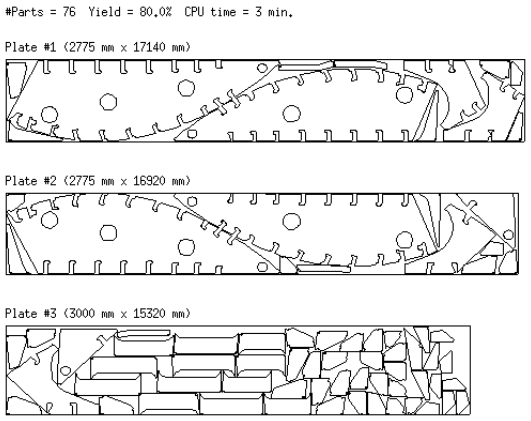


Fig. 11. Layout obtained for instance B by basic placement algorithm

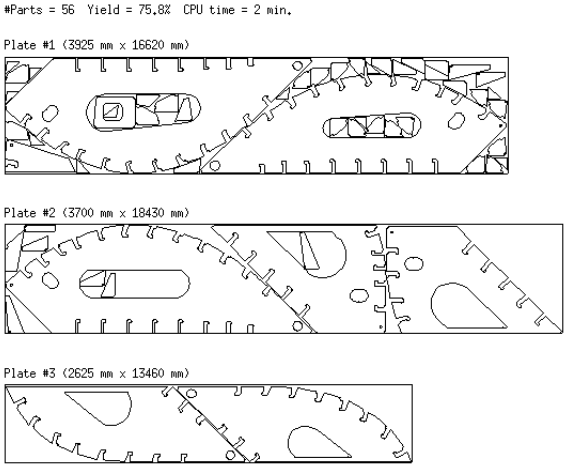


Fig. 12. Layout obtained for instance C by basic placement algorithm

5. Conclusion

A new approximation algorithm was proposed for the two-dimensional free-form bin packing (2D-FBP) problem. 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. The algorithm consists of a basic placement algorithm similar to the first-fit decreasing algorithm, which is known to be efficient for the one-dimensional case, and a group placement algorithm, in which combinations of placements are greedily examined. In this process, all the possible combinations of placements for a few consecutive items in the input items list are examined and the best placement is selected.

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 with those obtained by human experts, and the CPU times required by the algorithm are much faster than those required for manual nesting.

For future study, it is planned to improve the algorithm to represent both items and bins as sets of intervals, and compare its running time complexity with known theoretical results. Note that it is easy to modify the current program (*scan* and *place*1 shown in Appendix) to handle sets of intervals for both items and bins.

It is also 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.

Appendix

Figures 13 through 16 are pseudocode of the proposed basic placement algorithm. The parameter i specifies an item P_i to be placed, and parameters key and dir specify a placement. Key may be 1 or 2 if P_i has two baselines, and dir may be 1 or 2 that means the baseline should be facing up or down. In the pseudocode, for simplicity, key and dir is not referenced, and a placement with the baseline facing down is assumed.

$Place\_left$ finds the leftmost space in the current bin M into which the given item P_i may fit, and places 1's in the bitmap of M. $Place\_bottom$ finds the bottom-most space in M into which P_i may fit, and places 1's in the bitmap. To find empty spaces into which items may fit, they call *scan* subroutine, and to place 1's in the bitmap, they call *place*1 subroutine.

The notations used in figures are as follows:

P_i	An item to be placed,
l_i	Width of item P_i,
r_i	Number of scanlines in P_i for the selected placement,
$R_{ij}[s_{ij} \times 2 + 1]$	Run-length array of P_i where $j = 1, \ldots, r_i$,
s_{ij}	Number of {0,1}-intervals in R_{ij},
M	Current bin,
a	Width of bin M,
b	Number of scanlines in M,
$B_m[a]$	Bitmap arrays of M, where $m = 1, \ldots, b$.

```
1:  function place_left(i, key, dir : integer)          (* Place an item P_i in a current bin M *)
2:           : boolean;                                 (*     with the specified placement. *)
3:      var j, x, y, pos : integer
4:          next : array [1..b − r_i + 1] of integer    (* Array of x positions in M. *)
5:      begin
6:      for j := 1 to b − r_i + 1 do next[j] := 1;      (* Initialize next[ ]. *)
7:      while true do                                   (* Main loop. *)
8:          begin
9:          x := a;                                     (* Set x a large value (width of M). *)
10:         for j := 1 to b − r_i + 1 do                (* Find the smallest value in next[ ], *)
11:             if x > next[j] then                     (*     which is the leftmost position *)
12:                 begin x := next[j]; y := j; end;    (*     of empty area. *)
13:         if x > a − l_i + 1 then                     (* If there is no space where P_i will fit, *)
14:             begin place_left :=false; goto 22 end;  (*     return false. *)
15:         pos := scan(i, x, y);                       (* Check if P_i fits at position (x, y). *)
16:         if pos = 0 then                             (* If P_i fits at (x, y), *)
17:             begin place1(i, x, y);                  (*     place it there, *)
18:                   place_left :=true;                (*     and return true. *)
19:                   goto 22 end;
20:         next[y] := pos;                             (* Save the position. *)
21:         end;
22:     end;
```

Fig. 13. Pseudocode for placing an item at the leftmost position in a bin

```
1:  function place_bottom(i, key, dir : integer)        (* Place an item P_i in a current bin M *)
2:           : boolean;                                 (*     with the specified placement. *)
3:      var x, y, pos : integer
4:      begin
5:      for y := 1 to b − r_i + 1 do                    (* Scan M from the bottom to the top. *)
6:          begin
7:          x := 1;
8:          while true do                               (* Scan M from the left to the right. *)
9:              begin
10:             pos := scan(i, x, y);                   (* Check if P_i fits at position (x, y). *)
11:             if pos = 0 then                         (* If P_i fits at (x, y), *)
12:                 begin place1(i, x, y);              (*     place it there, *)
13:                       place_bottom :=true;          (*     and return true. *)
14:                       goto 19 end;
15:             x := pos;                               (* The next x position. *)
16:             end;
17:         end;
18:     place_bottom :=false;                           (* There is no space where P_i will fit. *)
19:     end;
```

Fig. 14. Pseudocode for placing an item at the bottom-most position in a bin

```
1:  function scan(i, x, y : integer) : integer;        (* Check if P_i fits at (x, y) in M. *)
2:      var j, k, o, p, pos, sum : integer;
3:      begin
4:      for j := 1 to r_i do                          (* Scan P_i from the bottom to the top. *)
5:          begin
6:          pos := x;                                 (* Positions from the left end of M *)
7:          sum := 0;                                 (*      and P_i, respectively. *)
8:          for k := 0 to s_ij - 1 do                 (* Scan P_i from the left to the right. *)
9:              begin
10:             pos := pos + R_ij[k × 2 + 1];         (* Move to the right *)
11:             sum := sum + R_ij[k × 2 + 1];         (*      by the length of one 0-interval. *)
12:             for o := 1 to R_ij[k × 2 + 2] do      (* Repeat by the length of one 1-interval. *)
13:                 begin
14:                 if not B_{y+j-1}[pos] = 0 then    (* If P_i does not fit at (x, y), find the *)
15:                     begin                         (*      next position where P_i may fit. *)
16:                     for p := pos to a - l_i + 1 do
17:                         if B_{y+j-1}[p] = 0 then goto 18
18:                     scan := p - sum;              (* Return the promising x position *)
19:                     goto 27                       (*      to use later in the caller. *)
20:                     end;
21:                 pos := pos + 1;                   (* Move to the right by one dot. *)
22:                 end;
23:             sum := sum + R_ij[k × 2 + 2];         (* Move to the right *)
24:             end;                                  (*      by the length of one 1-interval. *)
25:         end;
26:     scan := 0;                                    (* If P_i fits at (x, y), return zero. *)
27:     end;
```

Fig. 15. Pseudo code for searching for a space into which the item will fit

```
1:  procedure place1(i, x, y : integer)               (* Set P_i at (x, y) in M. *)
2:      var j, k, o, pos : integer;
3:      begin
4:      for j := 1 to r_i do                          (* Scan P_i from the bottom to the top. *)
5:          begin
6:          pos := x;                                 (* A position from the left end of M. *)
7:          for k := 0 to s_ij - 1 do                 (* Scan P_i from the left to the right. *)
8:              begin                                 (* Move to the right *)
9:              pos := pos + R_ij[k × 2 + 1];         (*      by the length of one 0-interval. *)
10:             for o := 1 to R_ij[k × 2 + 2] do      (* Repeat by the length of one 1-interval. *)
11:                 begin
12:                 B_{y+j-1}[pos] := 1;              (* Place 1 in the bitmap *)
13:                 pos := pos + 1;                   (* Move to the right by one dot. *)
14:                 end;
15:             end;
16:         end;
17:     end;
```

Fig. 16. Pseudocode for placing 1's in the bitmap

References

[1] 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.

[2] 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, pp. 46-93, 1997.

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

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

[5] A. Albano and G. Sapuppo, "Optimal allocation of two-dimensional irregular shapes using heuristic search methods," *IEEE Trans. Systems Man Cybernetics*, SMC-10, 5, pp. 242-248, 1980.

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

[7] V.J. Milenkovic, "Rotational polygon containment and minimum enclosure," In *Proc. 14th ACM Symp. Computational Geometry*, pp. 1-8, 1998.

[8] W. Qu and J.L. Sanders, "A nesting algorithm for irregular parts and factors affecting trim losses," *Int. J. Prod. Res.*, 25, 3, pp. 381-397, 1987.

[9] S. Yamauchi and K. Tezuka, "Automatic Nesting System by Use of Genetic Algorithm," *J. Soc. of Naval Arch. of Japan*, 178, 5-21, pp. 707-712, 1995. (in Japanese)

[10] K. Ratanapan and C.H. Dagli, "An Object-Based Evolutionary Algorithm: The Nesting Solution," In *Proc. 1998 IEEE Conf. Evolutionary Computation*, pp. 581-586, 1998.

[11] C.H. Cheng, B.R. Feiring, and T.C.E. Cheng, "The cutting stock problem – a survey," *Int. J. Prod. Eno.*, 26, pp. 291-305, 1994.