MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Hybrid Metaheuristics for Packing Problems

Toshihide IBARAKI, Shinji IMAHORI
and Mutsunori YAGIURA

METR 2007–01 January 2007

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

The METR technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

Hybrid Metaheuristics for Packing Problems

Toshihide Ibaraki[1], Shinji Imahori[2] and Mutsunori Yagiura[3]

[1] Kwansei Gakuin University ibaraki@ksc.kwansei.ac.jp
[2] University of Tokyo imahori@simplex.t.u-tokyo.ac.jp
[3] Nagoya University yagiura@nagoya-u.jp

1 Introduction

We consider in this chapter the *two-dimensional packing problem* that asks to pack a given set of *items* into a given *container* without mutual overlap. There are many variants of this problem depending upon whether the items are rectangles or have irregular shapes, and how minimization of the container is defined.

Most of these variants are NP-hard, since they contain as a special case the bin packing problem, which is already known to be NP-hard. Local search and metaheuristic algorithms have been playing major roles in obtaining good approximate solutions for practical uses. We observe that many of such algorithms contain subproblems that ask to pack given items in an optimal manner under certain constraints, and such subproblems are solvable by techniques known as dynamic, linear and nonlinear programming. Thanks to the recent progress of mathematical programming, efficient softwares are available for the cases of linear and nonlinear programming. The resulting algorithms are *hybrid metaheuristics* in the sense that they are combinations of metaheuristics and mathematical programming.

In this chapter, we deal with the following types of the packing problem:

(a) Items are rectangles, where the size (i.e., *width* and *height*) of each rectangle is fixed in advance.
(b) Items are soft rectangles, whose sizes can be adjusted.
(c) Items have irregular shapes, which may be neither rectangular nor convex.

In all these problems, the container is assumed to be a rectangle with width W and height H, and its size is minimized in the sense of WH (area), $W+H$ (perimeter) or H (height) while fixing its width to $W = W^*$. The last type is called the problem of *strip packing*.

We describe that dynamic programming can be effectively used for solving (a), and nonlinear and linear programming techniques for solving (b) and (c).

We also touch upon the problem of packing *rectangles with weights* under the constraints on the location of the center of gravity and their moment. Nonlinear programming is also useful for such a variant.

The organization of this chapter is as follows. Section 2 gives basic definitions and presents metaheuristic frameworks for the above packing problems. It then specifically discusses problem (a), in which dynamic programming techniques are employed. Section 3 considers problem (b), in which linear programming and nonlinear programming are used. Section 4 briefly mentions that a similar method can be used to pack rectangles with weights. Finally, Section 5 deals with problem (c), where linear programming and nonlinear programming are again used. All sections are concluded with some computational results.

2 Rectangle Packing Problem

Packing a number of rectangles, each having a fixed size, is perhaps most popular among packing problems. It is encountered in many industrial applications, such as wood, glass and steel manufactures, LSI and VLSI design, and newspaper paging. As the problem is NP-hard, various approximation algorithms have been proposed [17, 36]. Metaheuristics have also been utilized [14, 28, 31, 42].

We mainly treat the strip packing problem in this section. It may have additional constraints concerning orientation of rectangles and guillotine cut restriction [20, 58]. As for the orientation of rectangles, the following three situations have been considered in the literature: (1) each rectangle can be rotated by any angle, (2) each rectangle can be rotated by 90 degrees, and (3) each rectangle has a fixed orientation. Rotation of rectangles is not allowed in newspaper paging or when the rectangles to be cut are decorated or corrugated, whereas orientation is allowed in the case of plain materials. The guillotine cut constraint signifies that the rectangles must be obtained through a sequence of edge-to-edge cuts parallel to the edges of the container, which is usually imposed by technical limitations of the automated cutting machines. In this section, we mainly focus on case (3) without the guillotine cut constraint.

2.1 Problem formulation

We are given n items $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ of rectangular shape, where each rectangle $I_i \in \mathcal{I}$ has fixed width w_i and height h_i. We are asked to pack all items orthogonally into the *strip* (container) of a fixed width $W = W^*$ and a variable height H so as to minimize H. “Orthogonally” means that an edge of each item is parallel to an edge of the strip.

We describe the location of an item I_i by the coordinate (x_i, y_i) of its bottom-left corner. The problem is formally described as follows.

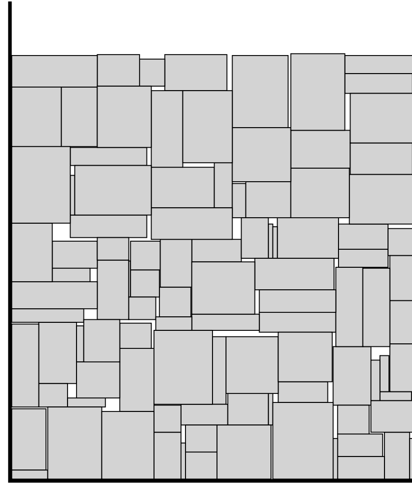


Fig. 1. An example of strip packing (rp100 with $W^* = 450$)

$$
\begin{array}{lll}
\text{minimize} & H & \\
\text{subject to} & 0 \le x_i \le W^* - w_i, \quad 1 \le i \le n & (1)\\
 & 0 \le y_i \le H - h_i, \quad 1 \le i \le n & (2)\\
 & \text{At least one of the next four inequalities} & \\
 & \text{holds for every pair } I_i \text{ and } I_j \text{ of rectangles:} & \\
 & x_i + w_i \le x_j, \qquad x_j + w_j \le x_i, & \\
 & y_i + h_i \le y_j, \qquad y_j + h_j \le y_i. & (3)
\end{array}
$$

The constraints (1) and (2) mean that every rectangle must be placed into the strip. The constraint (3) means that no two rectangles overlap; that is, each inequality signifies one of the four relative locations required to avoid mutual overlap: right-of, left-of, above and below.

We call a solution of the above problem (i.e., locations of all rectangles) as a *placement*. Figure 1 shows a placement obtained by the algorithm of this section, for the benchmark known as rp100.[4]

2.2 Coding schemes and decoding algorithms

In order to design algorithms for the rectangle packing problem, coding schemes and decoding algorithms should be discussed first.

In the rectangle packing problem, if we search the x and y coordinates of each rectangle directly, an effective search will be difficult because the number

[4] Available from `http://www.simplex.t.u-tokyo.ac.jp/~ imahori/packing/instance.html`

of solutions is uncountably many and elimination of mutual overlap is not easy. To overcome this difficulty, most algorithms for the rectangle packing problem are based on some coding schemes. A *coding scheme* consists of a set of coded solutions and a mapping from coded solutions to placements, and a *decoding algorithm* computes for a given coded solution the corresponding placement defined by the mapping. Desirable properties of a coding scheme and a decoding algorithm are summarized as follows.

1. There is a coded solution that corresponds to an optimal placement.
2. The number of all possible coded solutions is finite, where a smaller number is preferable provided that property 1 holds.
3. Every coded solution corresponds to a feasible placement.
4. A fast algorithm (running in polynomial time) for decoding is available.

A standard coding scheme is to represent a solution by a permutation of n rectangles, where the permutation specifies an order of placing rectangles one by one. The number of all permutations is $n!$, and every permutation corresponds to a placement without mutual overlap. In order to find a good permutation among them, heuristics and metaheuristics are used in almost all cases. A typical heuristics is just sorting the rectangles by some criteria; e.g., decreasing width, decreasing height, decreasing perimeter and decreasing area. In other cases, we search good permutations by local search or metaheuristics. The corresponding placements are computed by various decoding algorithms, which are also called placement rules; e.g., first fit [17], bottom left [6], and best fit [14] algorithms. We should note that, for a given permutation, different decoding algorithms may give different placements. In order to design a good packing algorithm using a permutation coding scheme, it is very important to choose a good decoding algorithm.

There are other types of coding schemes. All the schemes we explain hereafter specify relative positions between each pair of rectangles I_i and I_j (i.e., one of the four inequalities of (3)). The placement corresponding to a coded solution is the best one among those satisfying the relative positions specified by the coded solution.

Seqence pair Most well-known in this category is perhaps the *sequence pair* coding scheme [44]. A sequence pair is a pair of permutations $\sigma = (\sigma_+, \sigma_-)$ of $\{1, 2, \ldots, n\}$, where $\sigma_+(l) = i$ (equivalently $\sigma_+^{-1}(i) = l$) means that rectangle I_i is the lth rectangle in permutation σ_+. Permutation σ_- is similarly defined. A sequence pair $\sigma = (\sigma_+, \sigma_-)$ specifies which of the four conditions in (3) holds for each pair of I_i and I_j, based on the partial orders $\preceq_\sigma^x$ and $\preceq_\sigma^y$ defined by

$$\sigma_+^{-1}(i) \leq \sigma_+^{-1}(j) \text{ and } \sigma_-^{-1}(i) \leq \sigma_-^{-1}(j) \iff i \preceq_\sigma^x j,$$
$$\sigma_+^{-1}(i) \geq \sigma_+^{-1}(j) \text{ and } \sigma_-^{-1}(i) \leq \sigma_-^{-1}(j) \iff i \preceq_\sigma^y j.$$

This says that, if i appears before j in both σ_+ and σ_-, then $i \preceq_\sigma^x j$, i.e., I_i is placed to the left of I_j, while if i appears after j in σ_+, but before j in σ_-, then $i \preceq_\sigma^y j$, i.e., I_i is placed under I_j. Since exactly one of $i \preceq_\sigma^x j$, $j \preceq_\sigma^x$

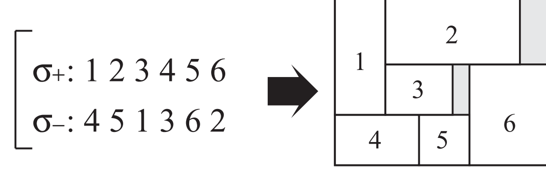


Fig. 2. A sequence pair $\sigma = (\sigma_+, \sigma_-)$ and its placement

i, $i \preceq^y_\sigma j$, $j \preceq^y_\sigma i$ always holds for a given pair of I_i and I_j, the constraints in (3) can be given by the following inequalities:

$$\begin{aligned} x_i + w_i \le x_j \quad \text{if} \quad i \preceq^x_\sigma j, \qquad x_j + w_j \le x_i \quad \text{if} \quad j \preceq^x_\sigma i, \\ y_i + h_i \le y_j \quad \text{if} \quad i \preceq^y_\sigma j, \qquad y_j + h_j \le y_i \quad \text{if} \quad j \preceq^y_\sigma i. \end{aligned} \tag{4}$$

Decoding algorithms Once we are given a sequence pair, we can compute a best placement satisfying the constraints (1), (2) and (4) in polynomial time using appropriate decoding algorithms; e.g., Murata et al. [44] proposed an $O(n^2)$ time decoding algorithm, Takahashi [55] improved it to $O(n \log n)$, Tang et al. [56] further improved it to $O(n \log \log n)$. Here, we briefly explain basic ideas in these algorithms. From the definition, we can obtain a feasible placement even if we compute the x and y coordinates separately; thus, we explain only the x direction. Let us define a set J_i for each rectangle I_i as follows:

$$J_i = \{ j \mid \sigma_+^{-1}(j) < \sigma_+^{-1}(i) \text{ and } \sigma_-^{-1}(j) < \sigma_-^{-1}(i) \}.$$

Then, the horizontal coordinates of each rectangle I_i can be computed by

$$x_i = \begin{cases} 0, & \text{if } J_i = \emptyset \\ \max_{j \in J_i} \{ x_j + w_j \}, & \text{otherwise.} \end{cases} \tag{5}$$

If we compute (5) naively for all i, it takes $O(n^2)$ time. But we can reduce this to $O(n \log n)$ by using a binary search tree as data structure.

Other coding schemes There are other coding schemes to specify one of the four relative locations for each pair of rectangles. We briefly explain some of them. One traditional coding scheme is to represent a solution by a binary tree of n leaves [52]. This coding scheme can represent only slicing structures; in other words, each placement obtained by this representation always satisfies the guillotine cut constraint. The leaves of a binary tree correspond to rectangles, and each internal node has a label 'h' or 'v', where h stands for horizontal and v stands for vertical. In this scheme, one of the four relative locations in (3) is assigned for each pair of rectangles as follows: Let u be the least common ancestor of I_i and I_j. If u has label 'h' and I_i is a left descendant of u (equivalently I_j is a right descendant of u), then we place I_i to the left of I_j. If the label of u is 'v', then we place I_i below I_j. When we are given

a binary tree, natural decoding algorithms can compute the best placement satisfying the above constraints in $O(n)$ time.

Guo et al. [26] and Chang et al. [15] also proposed coding schemes based on tree structures called O-tree and B*-tree, respectively. Their coding schemes can represent both of slicing and non-slicing structures, and compute a placement in $O(n)$ time.

Nakatake et al. [47] proposed another coding scheme called bounded sliceline grid (BSG in short). BSG has a grid structure and each pair of rooms in the grid represent one of the four relative positions. We assign all the rectangles to rooms, where at most one rectangle can be assigned to each room. It is argued that $O(n)$ number of rooms are sufficient in practice. Thus, given a coded solution (i.e., an assignment of rectangles to rooms), we can determine the corresponding placement in $O(n)$ time. A hybrid metaheuristic algorithm using BSG coding scheme was proposed by Imahori et al. [32].

2.3 Local search and simple metaheuristics

In this section, we explain the general idea of *local search* (LS in short). LS starts from an initial solution and repeats replacing the current solution with a better solution in its neighborhood until no better solution is found in the neighborhood. Here we focus on the LS for the strip packing problem of rectangles, which is based on sequence pairs, though it can be easily generalized to other settings.

Algorithm LS
Input: Data of widths w_i and heights h_i, $i = 1, 2, \ldots, n$, of rectangles and the width W^* of the container.
Output: A placement of all rectangles.

Step 1 (initialization): Construct an initial sequence pair $\sigma = (\sigma_+, \sigma_-)$. Compute the placement $\boldsymbol{v}(\sigma)$ and its objective value $z(\sigma)$ corresponding to σ, and let $\boldsymbol{v} := \boldsymbol{v}(\sigma)$ and $z := z(\sigma)$, where $\boldsymbol{v}$ and z denote the incumbent solution and its value, respectively.

Step 2 (local search): Repeat the following procedure until all sequence pairs in $N(\sigma)$ have been tested, where $N(\sigma)$ denotes the neighborhood of σ.

Select a new $\sigma' \in N(\sigma)$ and compute $\boldsymbol{v}(\sigma')$ and $z(\sigma')$. If $z(\sigma') < z$ holds, then let $\boldsymbol{v} := \boldsymbol{v}(\sigma')$, $z := z(\sigma')$, $\sigma := \sigma'$ and return to Step 2.

If there is no new sequence pair left in $N(\sigma)$, go to Step 3.

Step 3 (termination): Output $\boldsymbol{v}$, and halt.

The search space of LS is the set of sequence pairs, whose size is $(n!)^2$. An initial solution is often generated randomly; we generate two random permutations and use them as an initial solution. It is also possible to generate an initial solution using some heuristic algorithm. The quality of each solution

generated during search is evaluated by a given *evaluation function* $z(\sigma)$. It may be equal to the objective function or may be modified from it to make the search more effective.

Standard neighborhoods The solution output in Step 3 is *locally optimal* in neighborhood $N(\sigma)$. The performance of LS critically depends on how the neighborhood is designed. $N(\sigma)$ is commonly defined as the set of sequence pairs obtained from σ by applying certain local operations. Typical operations are *shift, swap* and *swap** [30, 31, 44], defined as follows.

1. Shift: This operation moves an element i in σ_+ (or σ_-) to the first position or to the next position of an element j. The shift neighborhood is defined by applying this to all pairs of i and j. If only one of σ_+ and σ_- is considered, it is the *single-shift neighborhood*, while if each shift operation is applied to both σ_+ and σ_-, then it is the *double-shift neighborhood*. The size of single-shift neighborhood is $O(n^2)$ since we consider all pairs of i and j. The size of double-shift neighborhood is $O(n^3)$ if we consider the j in σ_+ and σ_- independently. On the other hand, if we always select the same j in both σ_+ and σ_-, then the size becomes $O(n^2)$. In this case, we insert i before and after j, respectively, thereby examining four positions for each j. We call this the *limited double-shift neighborhood*.

2. Swap: This operation exchanges the positions of i and j in σ_+ (or in σ_-). The *single-swap neighborhood* and *double-swap neighborhood* are defined similarly to the case of shift neighborhood. The sizes of the resulting neighborhoods are $O(n^2)$.

3. Swap*: Let i and j in σ_+ satisfy $\sigma_+(\alpha) = i$ and $\sigma_+(\beta) = j$, with $\alpha < \beta$. Then for each γ with $\alpha \le \gamma < \beta$ we move i and j to location γ in the manner $\sigma'_+(\gamma) = j$ and $\sigma'_+(\gamma + 1) = i$, while keeping the same relative positions of other elements. This swap* operation can also be defined for σ_-. We usually apply swap* operations to only one of σ_+ and σ_-, yielding the *swap* neighborhood*. If we consider all combinations of i, j, γ, its size becomes $O(n^3)$.

The effects of these operations may be intuitively explained as follows, where we assume for simplicity that I_i is constrained to the left of I_j by σ. A shift operation applied to σ_+ changes the relative positions of I_i and I_j to "I_j above I_i", causing side effects on relative positions between I_i and other rectangles. A single-swap operation on σ_+ (resp., σ_-) changes the relative position "I_i to the left of I_j" to "I_j above I_i" (resp., "I_i above I_j"). On the other hand, a double-swap operation exchanges only the locations of I_i and I_j, without changing the relative positions of other rectangles. A swap* operation brings I_i and I_j together to their middle locations specified by γ.

Evaluation function In order to obtain $z(\sigma)$, we compute the placement of all rectangles by a decoding algorithm and use its objective value as $z(\sigma)$, for example. Here, we should note the following.

1. We denote

$$x_{\max} = \max_{I_i \in \mathcal{I}}\{x_i + w_i\} \quad \text{and} \quad y_{\max} = \max_{I_i \in \mathcal{I}}\{y_i + h_i\}.$$

   For some sequence pair, it may happen that $x_{\max} > W^*$ holds, that is, there exists a rectangle that protrudes from the strip width. In order to evaluate such infeasible solutions, we employ the following:

$$z(\sigma) = y_{\max} + M \times \max\{0,\ x_{\max} - W^*\}, \tag{6}$$

   where M is a large constant.
2. It may frequently happen that $z(\sigma') = z$ holds in Step 2 of LS. For an effective local search, we need some mechanisms to break such ties.

Metaheuristics In general, if LS is applied only once, many solutions of better quality may remain unvisited in the search space. Such phenomenon may be remedied by employing ideas of *metaheuristics*. We describe here two simple metaheuristic algorithms.

(1) The *random multi-start local search* (MLS). MLS randomly generates many initial solutions and apply LS to each initial solution independently. Then, the best of the obtained locally optimal solutions is output.

(2) The *iterated local search* (ILS) [35]. ILS is a variant of MLS, in which initial solutions are generated by slightly perturbing a good solution obtained during the search so far. In order to improve the performance of ILS, it is important to generate initial solutions which retain some features of good solutions and to avoid a cycling of solutions.

2.4 Hybrid metaheuristics for rectangle packing

We now describe a hybrid metaheuristic algorithm proposed by [31] for the rectangle packing problem, which is based on the sequence pair coding scheme. We first explain some ideas to decrease the neighborhood size, and then show efficient evaluation algorithms of dynamic programming.

Critical paths and neighborhood reductions As noted in Section 2.3, the size of the shift neighborhood is $O(n^2)$ or $O(n^3)$. In order to reduce this size without sacrificing its effectiveness, we restrict (1) the choice of element i that will be shifted in σ, and (2) the positions of j to where the i is inserted.

In order to restrict the rectangle I_i, we utilize critical paths. Critical paths are defined for both of the x (horizontal) and y (vertical) directions. We first consider the y direction. Given a placement, we define a directed graph $G = (V, E)$ and subsets $S, T \subseteq V$ as follows:

$$\begin{aligned}
&V = \{1, 2, \ldots, n\},\\
&(i, j) \in E \Longleftrightarrow i \preceq_{\sigma}^{y} j \ \text{ and } \quad y_i + h_i = y_j,\\
&S = \{i \mid y_i = 0\}, \quad T = \{i \mid y_i + h_i = y_{\max}\}.
\end{aligned}$$

Then, we define a *critical path* as a directed path in G, whose initial vertex s is in S and final vertex t is in T. For any placement obtained from a sequence pair σ, S and T are nonempty and there is at least one critical path. It is possible to find all rectangles on critical paths in $O(n)$ time. The definition for the x direction is similar except for the following situation: If $x_{\max} \leq W^*$, we need not to reduce $x_{\max}$, and hence we do not consider critical paths of x direction. To reduce the size of shift neighborhood, we shift only those rectangles I_i on critical paths. It is easy to show that a solution is locally optimal in the original shift neighborhood if no improved solution is found in the reduced neighborhood.

In order to reduce the size of neighborhood further, we consider only the following three types of neighborhoods with some restrictions.

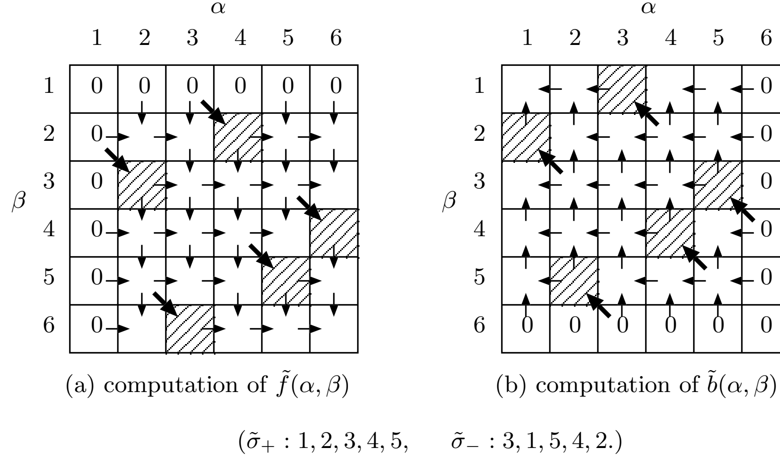
(1) The single-shift neighborhood, whose size is $O(n^2)$.
(2) The limited double-shift neighborhood, whose size is $O(n^2)$.
(3) As another reduced double-shift neighborhood, we insert i only to the positions close to the current position of i in σ_+ and σ_-, respectively. To control its size, we restrict the distance from the original position to the new positions within $a\sqrt{n}$, where a is a parameter. The size of this neighborhood is $O(a^2 n)$ for each i. We call this the *proximal double-shift neighborhood*.

Fast decoding by dynamic programming First we describe an algorithm to compute the $x_{\max}$ for all solutions in the double-shift neighborhood $N(\sigma)$. Assume that a rectangle I_i will be shifted. Here, we regard a shift operation as consecutive two operations: Deleting i from σ (we denote the resulting sequence pair by $\tilde{\sigma}$), and inserting i into other positions in $\tilde{\sigma}_+$ and $\tilde{\sigma}_-$ of $\tilde{\sigma}$. For the sequence pair $\tilde{\sigma}$, we compute the corresponding placement in $O(n \log n)$ time by a decoding algorithm noted in Section 2.2. Let $\tilde{x}_j$ be the x coordinate of rectangle I_j and $\tilde{x}_{\max} = \max_{I_j \in \mathcal{I} \setminus \{I_i\}} \{\tilde{x}_j + w_j\}$.

Now we insert i to the αth position in σ_+ and to the βth position in σ_-, respectively, and denote the resulting length of the critical path by $\tilde{x}_{\max}(\alpha, \beta)$. It is important to know whether I_i is on the horizontal critical path or not. If it is not on the critical path, then $\tilde{x}_{\max}(\alpha, \beta)$ is equal to $\tilde{x}_{\max}$. Otherwise we compute the length of the critical path that includes rectangle I_i by dynamic programming. Let us define $\tilde{J}^{\mathrm{f}}_{\alpha,\beta}$, $\tilde{J}^{\mathrm{b}}_{\alpha,\beta}$, $\tilde{f}(\alpha, \beta)$ and $\tilde{b}(\alpha, \beta)$ for each pair of α and β such that $1 \leq \alpha, \beta \leq n$ as follows:

$\tilde{J}^{\mathrm{f}}_{\alpha,\beta} = \{j \mid \tilde{\sigma}_+^{-1}(j) < \alpha,\ \tilde{\sigma}_-^{-1}(j) < \beta\}$,
$\tilde{J}^{\mathrm{b}}_{\alpha,\beta} = \{j \mid \tilde{\sigma}_+^{-1}(j) \geq \alpha,\ \tilde{\sigma}_-^{-1}(j) \geq \beta\}$,
$\tilde{f}(\alpha, \beta)$: length of the critical path for the set $\{I_j \mid j \in \tilde{J}^{\mathrm{f}}_{\alpha,\beta}\}$,
$\tilde{b}(\alpha, \beta)$: length of the critical path for the set $\{I_j \mid j \in \tilde{J}^{\mathrm{b}}_{\alpha,\beta}\}$.

Based on the idea of dynamic programming, $\tilde{f}(\alpha, \beta)$ (respectively, $\tilde{b}(\alpha, \beta)$) can be computed by



(a) computation of $\tilde{f}(\alpha, \beta)$ (b) computation of $\tilde{b}(\alpha, \beta)$

$(\tilde{\sigma}_+ : 1, 2, 3, 4, 5, \quad \tilde{\sigma}_- : 3, 1, 5, 4, 2.)$

Fig. 3. An example of computing $\tilde{f}(\alpha, \beta)$ and $\tilde{b}(\alpha, \beta)$

$$\tilde{f}(\alpha, \beta) = \begin{cases} 0, & \text{if } \alpha = 1 \text{ or } \beta = 1 \\ \max\{\tilde{f}(\alpha - 1, \beta), \tilde{f}(\alpha, \beta - 1)\}, & \text{if } \tilde{\sigma}_+(\alpha - 1) \neq \tilde{\sigma}_-(\beta - 1) \\ \tilde{f}(\alpha - 1, \beta - 1) + w_j, & \text{if } \tilde{\sigma}_+(\alpha - 1) = \tilde{\sigma}_-(\beta - 1) = j, \end{cases} \tag{7}$$

$$\tilde{b}(\alpha, \beta) = \begin{cases} 0, & \text{if } \alpha = n \text{ or } \beta = n \\ \max\{\tilde{b}(\alpha + 1, \beta), \tilde{b}(\alpha, \beta + 1)\}, & \text{if } \tilde{\sigma}_+(\alpha) \neq \tilde{\sigma}_-(\beta) \\ \tilde{b}(\alpha + 1, \beta + 1) + w_j, & \text{if } \tilde{\sigma}_+(\alpha) = \tilde{\sigma}_-(\beta) = j, \end{cases} \tag{8}$$

for all pairs of $\alpha = 1, 2, \ldots, n$ and $\beta = 1, 2, \ldots, n$ (resp., for all pairs of $\alpha = n, n-1, \ldots, 1$ and $\beta = n, n-1, \ldots, 1$). See Figure 3 as an example of showing the computation order of $\tilde{f}(\alpha, \beta)$ and $\tilde{b}(\alpha, \beta)$. Each box in this figure corresponds to $\tilde{f}(\alpha, \beta)$ (resp., $\tilde{b}(\alpha, \beta)$) for each pair of α and β, and arrows show how to compute each value. For the $\tilde{\sigma}_+$ and $\tilde{\sigma}_-$ given in the figure, the value of each shaded box is computed by the third formula of (7) (resp., (8)). For the new placement after inserting I_i into the αth position of $\tilde{\sigma}_+$ and the βth position of $\tilde{\sigma}_-$, we can compute the critical path length that includes rectangle I_i by $\tilde{f}(\alpha, \beta) + w_i + \tilde{b}(\alpha, \beta)$. Thus,

$$\tilde{x}_{\max}(\alpha, \beta) = \max\{\tilde{x}_{\max}, \tilde{f}(\alpha, \beta) + w_i + \tilde{b}(\alpha, \beta)\}. \tag{9}$$

This algorithm takes $O(n \log n)$ time for the original decoding algorithm applied to $\tilde{\mathcal{I}}$ and $\tilde{\sigma}$. Time to compute $\tilde{f}(\alpha, \beta)$ and $\tilde{b}(\alpha, \beta)$ for all pairs of α and β by (7) and (8) is $O(n^2)$. Time to compute $\tilde{x}_{\max}(\alpha, \beta)$ by (9) is $O(1)$ for each pair of α and β, and it becomes $O(n^2)$ for all pairs of α and β. In summary, the total computation time of this algorithm, i.e., time to evaluate all solutions when rectangle I_i is shifted in the double-shift neighborhood is $O(n^2)$. This implies that it takes $O(1)$ amortized time to evaluate one coded solution in the double-shift neighborhood.

We then consider the limited double-shift neighborhood. As in the above algorithm, we first delete i from σ and compute the placement for $\tilde{\sigma}$. In this case, we compute $\tilde{f}(\alpha, \beta)$ and $\tilde{b}(\alpha, \beta)$ only for necessary pairs of α and β. That is, when we insert i before or after $j = \tilde{\sigma}_+(\alpha - 1) = \tilde{\sigma}_-(\beta - 1)$ in $\tilde{\sigma}_+$ and $\tilde{\sigma}_-$, respectively, we only need to compute $\tilde{f}(\alpha-1, \beta-1)$, $\tilde{f}(\alpha-1, \beta)$, $\tilde{f}(\alpha, \beta-1)$ or $\tilde{f}(\alpha, \beta)$. However, these can be immediately given by $\tilde{f}(\alpha - 1, \beta - 1) = \tilde{f}(\alpha - 1, \beta) = \tilde{f}(\alpha, \beta - 1) = \tilde{x}_j$ and $\tilde{f}(\alpha, \beta) = \tilde{x}_j + w_j$. Thus the computation time for each solution is $O(1)$. The case of $\tilde{b}(\alpha, \beta)$ is similar.

To evaluate solutions obtainable in the single-shift neighborhood, where i is shifted in σ_+, we compute $\tilde{f}(\alpha, \beta)$ for all $1 \leq \alpha \leq n$ and $\beta = \sigma_-^{-1}(i)$ by

$$\tilde{f}(\alpha, \beta) = \begin{cases} \max\{\tilde{f}(\alpha - 1, \beta), \tilde{x}_{j'} + w_{j'}\}, & \text{if } \tilde{\sigma}_-^{-1}(j') \leq \beta - 2 \\ \tilde{x}_{j'} + w_{j'}, & \text{if } \tilde{\sigma}_-^{-1}(j') = \beta - 1 \\ \tilde{f}(\alpha - 1, \beta), & \text{otherwise,} \end{cases} \tag{10}$$

where $j' = \tilde{\sigma}_+(\alpha - 1)$. Similar formula can be derived if i is shifted in σ_-. In both cases, time to compute $\tilde{f}(\alpha, \beta)$ for all necessary α and β is $O(n)$.

To evaluate solutions in the proximal double shift neighborhood, we compute $\tilde{f}(\alpha, \beta)$ for all $\alpha_l \leq \alpha \leq \alpha_u$ and $\beta_l \leq \beta \leq \beta_u$, where $\alpha_u - \alpha_l \leq 2a\sqrt{n}$ and $\beta_u - \beta_l \leq 2a\sqrt{n}$ hold. We first compute $\tilde{f}(\alpha, \beta_l)$ for $1 \leq \alpha \leq \alpha_u$ by (10) and $\tilde{f}(\alpha_l, \beta)$ for $1 \leq \beta \leq \beta_u$ by the σ_- version of (10). Then, we use (7) to compute $\tilde{f}(\alpha, \beta)$ for all $\alpha_l + 1 \leq \alpha \leq \alpha_u$ and $\beta_l + 1 \leq \beta \leq \beta_u$. Therefore, we can compute $\tilde{f}(\alpha, \beta)$ for all necessary α and β in $O(a^2 n)$ time.

In summary, we can evaluate all solutions in the limited and proximal double-shift neighborhoods in $O(n \log n) + O(a^2 n)$ time for each shifted i. Thus the amortized computation time for one coded solution becomes $O(\log n)$.

2.5 Computational results

We give some computational results of heuristic, metaheuristic, and hybrid metaheuristic algorithms on test instances given by Hopper and Turton [28]. There are seven different categories called C1, C2, ..., C7 with the number of rectangles ranging from 17 to 197, where each category has three instances. We compare the following three algorithms: (1) A heuristic algorithm BLF-DW (bottom left fill with decreasing width) proposed by Baker et al. [6] and implemented by Hopper and Turton [28] (denoted BLFDW), (2) a simulated annealing algorithm with BLF algorithm by Hopper and Turton [28] (denoted SA-BLF) and (3) a hybrid metaheuristic algorithm based on ILS and dynamic programming by Imahori et al. [31] (denoted HM-SP). The results of algorithms BLFDW and SA-BLF are taken from [28], where these algorithms were coded in C++ language and run on a PC (Intel Pentium Pro 200 MHz, 65 MB memory). The results of HM-SP are taken from [31], which was coded in C language and run on a PC (Intel Pentium III 1 GHz, 1 GB memory). Based on the benchmark results of SPECint from SPEC web page (`http://www.specbench.org/`), the latter CPU is about six times faster than

Table 1. Comparison of three algorithms for the rectangle packing problem

		BLFDW		SA-BLF		HM-SP	
category	n	ratio	time	ratio	time	ratio	time
C1	17	89	< 0.1	96	42	97.56	10.0
C2	25	84	< 0.1	94	144	93.75	15.0
C3	29	88	< 0.1	95	240	96.67	20.0
C4	49	95	< 0.1	97	1980	96.88	150.0
C5	73	95	< 0.1	97	6900	97.02	500.0
C6	97	95	< 0.1	97	22920	96.85	1000.0
C7	197	95	0.64	96	250860	96.55	3600.0

the former. Results are shown in Table 1. Column “ratio” shows the average of the following ratio,

$$100 \times \frac{\text{(total area of rectangles)}}{\text{(output value of } H) \times W^*},$$

(i.e., the larger the better). Column “time” shows the computation time in seconds for one instance.

From the table, we observe that BLFDW is much faster than others, but the quality of output solutions is slightly worse. The solution quality of two metaheuristics are similar, but HM-SP is superior to SA-BLF in the computation time.

3 Packing Soft Rectangles

In this section we assume that all rectangles are soft. Namely, the width w_i and the height h_i of rectangle I_i can be adjusted within given constraints. For example, the constraints may specify their lower and upper bounds:

$$\begin{aligned} w_i^L &\le w_i \le w_i^U, \\ h_i^L &\le h_i \le h_i^U. \end{aligned} \tag{11}$$

We may also add the constraint that the *aspect ratio* h_i/w_i is bounded between its lower bound r_i^L and upper bound r_i^U:

$$r_i^L w_i \le h_i \le r_i^U w_i. \tag{12}$$

Another type of constraint common in applications is that each rectangle I_i must have either a given perimeter L_i or a given area A_i (or both):

$$w_i + h_i \ge L_i, \tag{13}$$

$$w_i h_i \ge A_i. \tag{14}$$

In addition, we may consider that the locations (x_i, y_i) of rectangles I_i are pre-determined in some intervals:

$$x_i^L \leq x_i \leq x_i^U, \\ y_i^L \leq y_i \leq y_i^U. \tag{15}$$

To our knowledge, there is not much literature on the problem using soft rectangles, except such papers as [16, 34, 45, 59] containing algorithms and [46] containing a theoretical analysis, even though the problem has wide applications.

Applications can be found, for example, in VLSI floorplan design [16, 34, 44, 45, 59] and in resource constrained scheduling. In the VLSI design, each rectangle represents a block of logic circuits consisting of a certain number of transistors, which occupy certain area, and must have at least some perimeter length to accommodate connection lines to other blocks. The shape of each rectangle is adjustable, but required to satisfy the constraints as stated above. In a scheduling application, each rectangle may represent a job, to be assigned to an appropriate position on the time axis (horizontal), where its width gives the processing time of the job and its height represents the amount of resource (per unit time) invested to process the job. In this case, the area of the rectangle represents the total amount of resource consumed by the job, which is again required to satisfy the above constraints.

In the following, we focus on the local search algorithm proposed by [29], which is based on the sequence pair (defined in Section 2.2). Given a sequence pair, the problem of computing the sizes and placement of rectangles is formulated as a linear programming problem or nonlinear programming problem (more exactly, convex programming problem) depending on the constraints and objective functions. Although these mathematical programming algorithms are quite efficient, it still consumes some amount of time, and it is carefully considered how to reduce the neighborhood sizes.

3.1 Problem statement

As the objective function, we may choose to minimize the perimeter of the container, i.e.,

$$\text{minimize} \quad W + H \tag{16}$$

or to minimize its area,

$$\text{minimize} \quad WH, \tag{17}$$

where W and H are the variables that satisfy

$$x_i + w_i \leq W \quad \text{for all } i, \\ y_i + h_i \leq H \quad \text{for all } i. \tag{18}$$

In the strip packing problem, the width of the container is fixed to $W = W^*$, and its height H is minimized:

$$\text{minimize} \quad H, \tag{19}$$

under the constraint

$$\begin{aligned} x_i + w_i &\leq W^* \quad \text{for all} \ \ i, \\ y_i + h_i &\leq H \quad\ \ \text{for all} \ \ i. \end{aligned} \tag{20}$$

Therefore, if a sequence pair σ is specified, we are required to solve the mathematical programming problem $P(\sigma)$ to minimize objective function (16), (17) or (19) under the constraints:

$$\begin{aligned} &(11), (12), (13) \text{ (and/or (14))}, \ (15), \ (18) \text{ (or (20))} \ \ \text{for all} \ \ i, \\ &(4) \qquad\qquad\qquad\quad \text{for all} \ \ i \text{ and } j, \\ &x_i, \ y_i \geq 0 \qquad\qquad \text{for all} \ \ i. \end{aligned} \tag{21}$$

It is important to note that the feasible region defined by constraints (21) is *convex*, as illustrated in Figure 4. Therefore, if the objective function is convex, we obtain a convex programming problem, and can solve it by existing efficient algorithms (e.g., [11, 48]). This is the case when we want to minimize (16) or (19). The problem with objective function (17) is not a convex programming problem, but is a so-called multiplicative programming problem for which some efficient approaches are also known (e.g., [38]).

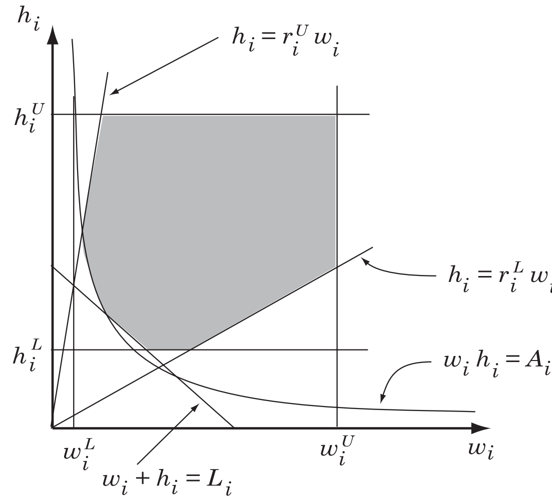


Fig. 4. Feasible region for the w_i and h_i of rectangle I_i

Two test problems From the above varieties, two simple problems will be discussed in the following. The first type minimizes the perimeter of the container under the perimeter constraints (13) of rectangles.

$$
\begin{array}{llll}
P_{\mathrm{peri}}(\sigma): \text{minimize} & W + H & & \\
\text{subject to} & (11), (13), (18) & \text{for all } i & \\
& (4) & \text{for all } i \text{ and } j & \quad (22) \\
& x_i,\ y_i \geq 0 & \text{for all } i. &
\end{array}
$$

This gives rise to a linear programming problem for each given sequence pair σ, and is called the *perimeter minimization problem.*

The second type is the strip packing problem under the area constraints (14) of rectangles, which is formulated as a convex programming problem.

$$
\begin{array}{llll}
P_{\mathrm{area}}(\sigma): \text{minimize} & H + Ms & & \\
\text{subject to} & (11), (14) & \text{for all } i & \\
& x_i + w_i \leq W^* + s & \text{for all } i & \\
& y_i + h_i \leq H & \text{for all } i & \\
& (4) & \text{for all } i \text{ and } j & \quad (23) \\
& s \geq 0 & & \\
& x_i,\ y_i \geq 0 & \text{for all } i. &
\end{array}
$$

Here the variable s is introduced to keep the problem feasible, by adding penalty term Ms to the objective function (19) with a large positive constant M. This is called the *area minimization problem.*

3.2 Neighborhood reductions

Starting from the standard neighborhoods as described in Section 2.3, we consider how to reduce their sizes further.

Critical paths Given a placement $\boldsymbol{v}(\sigma)$, we consider horizontal and vertical critical paths as described in Section 2.4. It is often attempted (e.g., see Section 2.4 and [30]) to restrict I_i to be in a critical path, while I_j can be any. We call the resulting neighborhoods like *single-swap critical neighborhood, swap* critical neighborhood* and so forth.

In handling soft rectangles, a placement $\boldsymbol{v}(\sigma)$ tends to have many critical paths, since each rectangle is adjusted so that it directly touches horizontally adjacent rectangles or vertically adjacent rectangles. As a result, the restriction to critical paths is not very effective in reducing the neighborhood size. To remedy this to some extent, we define the *single-swap lower-bounding critical neighborhood* by restricting I_i to be in a critical path and to satisfy $w_i = w_i^L$ if the critical path is horizontal (or $h_i = h_i^L$ if vertical), since such a rectangle I_i cannot be shrunk any further. Similar argument applies also to other types of neighborhoods, resulting in the *single-shift lower-bounding critical neighborhood* and others.

Computational comparison of neighborhoods To evaluate the power of the above neighborhoods, preliminary computational experiment was conducted in [29] for the following neighborhoods, abbreviated as

Sg-shift: single-shift neighborhood,
Db-shift: double-shift neighborhood,
Sg-swap: singel-swap neighborhood,
Db-swap: double-swap neighborhood,
SgCr-shift: single-shift critical neighborhood (similarly for DbCr-shift, SgCr-swap, DbCr-swap),
swap*: swap* neighborhood,
SgLb-shift: single-shift lower-bounding critical neighborhood (similarly for SgLb-swap and Lb-swap*),
SgAd-swap: single-swap adjacent lower-bounding critical neighborhood.

According to the computational results, SgLb-swap appears to be reasonably stable and gives good results in most cases. However, it still requires rather large computation time. To shorten its time, we further restrict rectangles I_i and I_j to be swapped to those which are lower bounding (i.e., $w_i = w_i^L$ or $h_i = h_i^L$ holds depending on the direction of the critical path) and are adjacent in some critical path. The resulting neighborhood is denoted as SgAd-swap. The quality of the solutions obtained by SgAd-swap is not good, but it consumes very little time compared with others.

Further elaborations To reduce the neighborhood sizes further while maintaining high searching power, three more modifications were added.

The first idea is to look at a rectangle which belongs to both horizontal and vertical critical paths. We call such a rectangle as a *junction* rectangle. It is expected that removing a junction rectangle will break both the horizontal and vertical critical paths, and will have a large effect of changing the current placement. Thus we apply single-shift or double-swap operations to a junction rectangle I_i with any other rectangles I_j which are not junctions (in the case of double-swap we further restrict I_j to have a smaller area than I_i). We then apply these operations to all junction rectangles I_i. If an improvement is attained in this process, we immediately move to local search with SgLb-swap neighborhood for attaining further improvement. This cycle of "junction removals" and "local search with SgLb-swap" is repeated until no further improvement is attained. The resulting algorithms are denoted Jc(Sg-shift)+SgLb-swap or Jc(Db-swap)+SgLb-swap, respectively, depending on which operation is used to move the junction rectangle.

To improve the efficiency further, it was tried to replace the SgLb-swap neighborhood in the above iterations with SgAd-swap, which was defined at the end of the previous subsection. Using this neighborhood in place of SgLb-swap, we obtain algorithms Jc(Sg-shift)+SgAd-swap or Jc(Db-swap)+SgAd-swap.

Experiment shows that these four attain similar quality, but the last one Jc(Db-swap)+SgAd-swap consumes much less computation time than others.

The last idea is to make use of vacant areas existing in a given placement. To find some of such vacant areas by a simple computation, we use the fol-

lowing property. Let the current sequence pair σ satisfy $i \preceq_\sigma^x j$ and there is no k such that $i \preceq_\sigma^x k \preceq_\sigma^k j$ (i.e., i is immediately to the left of j). In this case, if $x_i + w_i < x_j$ holds, there is some vacant area between i and j. We pick up the largest one among such vacant areas, in the sense of maximizing $x_j - (x_i + w_i)$. Let i^* and j^* be the resulting pair. Then we apply Sg-swap operations on σ^+ between those i and j such that $i \in \sigma^+$ is located in distance at most 5 from i^* (forward or backward, i.e., $|\sigma_+^{-1}(i) - \sigma_+^{-1}(i^*)| \leq 5$), and $j \in \sigma^+$ is located in distance at most 5 from j^* (forward or backward).

This neighborhood is derived by horizontal argument. Analogous argument can also be applied vertically, and we consider the local search based on the resulting two types of neighborhoods, denoted SgVc-swap.

As a conclusion of preliminary experiment, the following combined neighborhood was chosen.

Neighborhood A: Neighborhood Jc(Db-swap)+SgAd-swap with the addition of neighborhood SgVc-swap.

In the experiment of the next section, the two neighborhoods in A are combined in the following manner: First apply local search with Jc(Db-swap)+SgAd-swap until a locally optimal solution is obtained, and then improve it by local search with SgVc-swap. The best solution obtained is then output.

3.3 Computational results

Benchmarks and experiment The benchmarks[5] known as ami49 and rp100 were used. They involve 49 and 100 hard rectangles, respectively, whose widths and heights are denoted w_i^0 and h_i^0. In the case of the perimeter minimization problem, we set the lower and upper bounds on widths and heights as follows.

$$\begin{aligned} w_i^L &= (1-e)w_i^0,\ w_i^U = (1+e)w_i^0 \\ h_i^L &= (1-e)h_i^0,\ h_i^U = (1+e)h_i^0, \end{aligned} \tag{24}$$

where e is a constant like $0.1, 0.2$, etc. The perimeter L_i in (13) of each rectangle I_i is set to $L_i = w_i^0 + h_i^0$.

For the area minimization problem, we first set the areas A_i in constraint (14) by $A_i = w_i^0 h_i^0$ for all i, the bounds on h_i as in (24), and then the bounds on w_i by

$$w_i^L = A_i / h_i^U \quad \text{and} \quad w_i^U = A_i / h_i^L. \tag{25}$$

The algorithm was coded in C language, and run on a PC using Pentium 4 CPU, whose clock is 2.60 GHz and memory size is 780 MB. The linear and convex programming problems are solved by a proprietary software package

[5] See the footnote in Section 2.1

NUOPT[6] of Mathematical Systems Inc., where the linear programming is based on the simplex method and the convex programming is based on the line search method.

Perimeter minimization The first set of instances of the perimeter minimization problem (22) are generated from ami49 by setting constants e in (24) to $e = 0.0,\ 0.1,\ 0.2,\ 0.3$, respectively. For each e, five runs are conducted from independent random initial solutions and average data are given in Table 2. The meaning of rows is as follows: Candidates: The number of solutions σ tested, Improvements: The number of improvements attained in LS, Time: CPU time in seconds, Density: Total area of rectangles over the area of container, $W + H$ and H: Objective values. Note that Density and H are given both the average and best values in five runs.

From these results we see that the local search could obtain reasonably good solutions, except for the case of $e = 0.0$ (i.e., all rectangles are hard). As we reduced the neighborhood size to a great extent, in order to make the whole computation time acceptable, the resulting size appears not sufficient for handling hard rectangles.

Table 2. Perimeter minimization problem with different e

Benchmarks		$e = 0.0$	$e = 0.1$	$e = 0.2$	$e = 0.3$
	Candidates	413.8	7877.6	12400.0	11472.2
	Improvements	34.6	153.0	218.8	236.2
	Time (secs)	28.0	641.8	1142.4	897.4
ami49	Density(avg.)	61.3	92.6	97.1	97.6
	W+H(avg.)	15268.4	12346.3	11719.9	11401.9
	Density(best)	66.3	97.7	98.5	98.7
	W+H(best)	14644.0	11903.5	11686.8	11677.9

Table 3. Area minimization problem with fixed widths W^*

Benchmarks		$W^* = 800$	$W^* = 1000$	$W^* = 1200$
	Candidates	4156.2	3200.6	2195.0
	Improvements	241.2	184.2	140.4
	Time (secs)	967.2	743.2	511.6
ami49	Density(avg.)	98.8	95.9	91.9
	H (avg.)	8152.5	6744.5	5868.5
	Density(best)	99.5	99.5	98.5
	H (best)	8097.6	6479.5	5454.4

Area minimization The area minimization problem (23) was solved for three different W^* and $e = 0.2$, where five runs from random initial solutions were again carried out. The results are shown in Table 3. Although, in this

[6] `http://www.msi.co.jp/english/`

case, the convex programming problems $P_{\text{area}}(\sigma)$ are used instead of the linear programming problems, the computation time does not increase much, and very dense placements are obtained in most of the tested instances.

Finally, a large benchmark rp100 with 100 rectangles was tested. Table 4 gives the results of problems (22) and (23) with $e = 0.2$ and $W^* = 450$ (in the case of (23)). The obtained result for the area minimization is shown in Figure 5. Considering that 2–3 hours were consumed for each run, it appears difficult to handle larger instances than these with this approach.

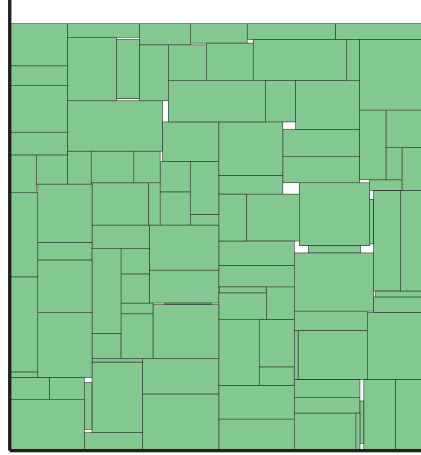


Fig. 5. Area minimization with rp100 ($W^* = 450$)

Table 4. Results with 100 rectangles

Benchmarks		Perimeter	Area
	Candidates	35012	15933
	Improvements	536	500
rp100	Time (secs)	7268.5	10101.5
	Density(%)	97.6	98.8
	H	888.8	461.4

4 Rectangles with Weights

The packing problem of rectangles with weights is also found in some applications, where each rectangle has weight d_i. In this case, the center of each rectangle is given by $(x_i + w_i/2,\ y_i + h_i/2)$, and constraints involving their center of gravity

$$(x^c, y^c) = \left(\frac{1}{D} \sum_i d_i (x_i + w_i/2),\ \frac{1}{D} \sum_i d_i (y_i + h_i/2) \right),$$

where $D = \sum_i d_i$, may be added. The objective function to minimize may be their kth moment (e.g., $k = 1$ or 2) around the center of gravity,

$$\sum_i d_i \left(\sqrt{(x_i + w_i/2 - x^c)^2 + (y_i + h_i/2 - y^c)^2} \right)^k .$$

This type of problem can also be handled in the local search framework as described in the previous section, in which the placement corresponding to a given sequence pair can be computed by nonlinear programming. An attempt in this direction is being made by [39]. Figure 6 shows a solution obtained in their preliminary experiment, where the center of gravity is constrained to be the middle and the first moment is minimized. Note that darker rectangles represent heavier items.

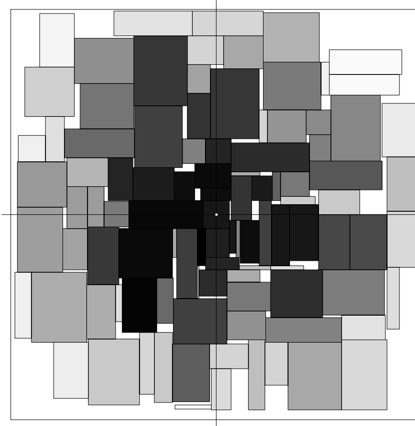


Fig. 6. Minimization of the first moment of rectangles with weights

5 Irregular Packing

In this section, we assume that given items are general polygons or arbitrary shapes that are not necessarily rectangular nor convex. Such problems are called *irregular packing* or *nesting* problems (or sometimes called *marker making*). Figure 7 shows an example of packing nonconvex items for the benchmark known as “swim” (see Section 5.7). There are many practical applications, e.g., in the garment, shoe and ship building industries, and many variants have been considered in the literature. Among them, the *irregular strip packing* problem has been extensively studied. Given n polygons and a rectangular

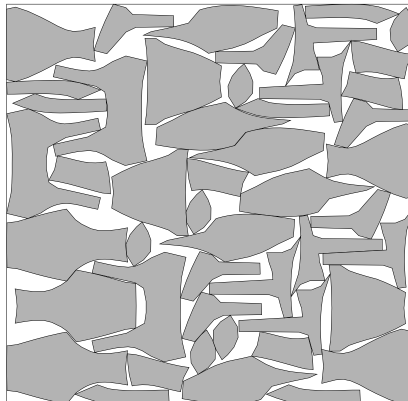


Fig. 7. An example of packing nonconvex polygons

container (i.e., a *strip*) with constant width W^* and variable height H, this problem asks to find a feasible placement of the given polygons into the strip so as to minimize the height H of the strip. As in the case of the rectangle packing problem, a placement is feasible if no polygon overlaps with any other polygon or protrudes from the strip. It has three variations with respect to the rotations of polygons: (1) rotations of any angles are allowed, (2) finite number of angles are allowed, and (3) no rotation is allowed. (Note that case (3) is a special case of (2) in which the number of given orientations of each polygon is one.) In this section, we mainly focus on case (3) for simplicity. In many practical applications such as textile industry, rotations are usually restricted to 180 degrees because textiles have the grain and may have a drawing pattern, while in such applications as glass, plastic etc., rotations of any angles are allowed. As mentioned in [27, 43], small rotations of any angles (e.g., a few degrees) in addition to 180 degrees are sometimes considered even for textiles in order to make the placement efficient.

A big difference of irregular packing from rectangle packing is that the intersection test between polygons is considerably more complex. Some heuristic algorithms use approximation of the given shapes, while many of the recent algorithms use a geometric technique called *no-fit polygons*, whose details will be explained in Section 5.2.

A standard way of designing heuristics is to put polygons one by one into the container according to a sequence of the given polygons. Most of the construction heuristics are based on this scheme, e.g., [3, 12, 51], which can be viewed as the irregular counterpart of the bottom-left heuristics for the rectangular case. A recent heuristic algorithm by Burke et al. [13] is quick and its solution quality is promising. It is also effective to apply local search (or metaheuristics) to find good sequences of polygons that lead to good placements when a construction algorithm is applied (i.e., a sequence is a coded solution and the construction algorithm is a decoding algorithm) [24].

Among many heuristic and metaheuristic algorithms, we focus our attention on hybrid approaches with mathematical programming. Such approaches seem to be effective in obtaining solutions of high quality especially when we have sufficient computation time.

In the following, we first define the irregular strip packing problem in Section 5.1, and then explain the idea of no-fit polygons in Section 5.2. In Section 5.3 we define some important subproblems, which are useful to solve the original packing problem efficiently. Then in Sections 5.4 and 5.5 we describe how mathematical programming techniques are utilized for the irregular packing problems. In Section 5.6 we briefly summarize metaheuristic algorithms incorporated with mathematical programming techniques, and finally in Section 5.7 we give some computational results of recent hybrid metaheuristics.

5.1 Irregular strip packing problem

We are given a set $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ of polygons, and a rectangular container (i.e., strip) $C = C(W^*, H)$ with a width $W^* \geq 0$ and a height H, where W^* is a constant and H is a nonnegative variable. We describe the location of a polygon P_i by the coordinate $\boldsymbol{v}_i = (x_i, y_i)$ of its *reference point*, where the reference point is any point of the polygon (e.g., a vertex of the polygon or the center of gravity; in the rectangular case, we set the bottom-left corner to be the reference point). The vector $\boldsymbol{v}_i = (x_i, y_i)$ is called the *translation vector* for P_i. For convenience, we regard each polygon P_i and the container C as the set of points inside it including the points on the boundary when its reference point is put at the origin $O = (0, 0)$. Then, we describe the polygon P_i placed at $\boldsymbol{v}_i$ by the Minkowski sum $P_i \oplus \boldsymbol{v}_i = \{\boldsymbol{p} + \boldsymbol{v}_i \mid \boldsymbol{p} \in P_i\}$. For a polygon S, let $\mathrm{int}(S)$ be the interior of S, ∂S be the boundary of S, $\bar{S}$ be the complement of S, and $\mathrm{cl}(S)$ be the closure of S. Then the irregular strip packing problem (ISP) is formally described as follows.

$$
\begin{array}{llll}
\text{(ISP)} & \text{minimize} & H & \\
 & \text{subject to} & \mathrm{int}(P_i \oplus \boldsymbol{v}_i) \cap (P_j \oplus \boldsymbol{v}_j) = \emptyset, & 1 \leq i < j \leq n \\
 & & (P_i \oplus \boldsymbol{v}_i) \subseteq C(W^*, H), & 1 \leq i \leq n \\
 & & H \geq 0, & \\
 & & \boldsymbol{v}_i \in \mathbf{R}^2, & 1 \leq i \leq n.
\end{array}
$$

We represent a solution of problem (ISP) by an n-tuple $\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n)$, which is the essential part of the decision variables because the minimum height H of the container is determined by

$$
\begin{aligned}
H_{\min}(\boldsymbol{v}) = & \max\{y \mid (x, y) \in P_i \oplus \boldsymbol{v}_i,\ P_i \in \mathcal{P}\} \\
& - \min\{y \mid (x, y) \in P_i \oplus \boldsymbol{v}_i,\ P_i \in \mathcal{P}\}
\end{aligned}
$$

and $(P_i \oplus \boldsymbol{v}_i) \subseteq C(W^*, H_{\min}(\boldsymbol{v}))$ holds for all $P_i \in \mathcal{P}$ if and only if

$$
\begin{aligned}
W^* \geq & \max\{x \mid (x, y) \in P_i \oplus \boldsymbol{v}_i,\ P_i \in \mathcal{P}\} \\
& - \min\{x \mid (x, y) \in P_i \oplus \boldsymbol{v}_i,\ P_i \in \mathcal{P}\}.
\end{aligned}
$$

5.2 Intersection test and no-fit polygon

We consider in this section how to test the intersection between polygons. One popular idea for speeding up this test is to represent the polygons approximately. Some heuristic algorithms [5, 50] are based on *raster* (or bitmap) representation of the given polygons. Main drawback of this approach is that an appropriate choice of raster size is not easy. If the raster is rough, the intersection test is quick, but it will suffer from inaccuracy caused by the approximation inherent in the raster representation. On the other hand, if the raster is minute, the intersection test becomes expensive, and the memory space to keep the raster representation of polygons becomes huge.

Okano [49] proposed a technique that approximates polygons by a set of parallel line segments, which is called *scanline* representation. (His algorithm was designed for the two-dimensional bin packing problem, but it is applicable to various irregular packing problems including (ISP).) The number of scanlines is usually much smaller than that of pixels in a raster representation when the same resolution is required.

One of the most popular geometric techniques used for the intersection test is *no-fit polygon*. This concept was introduced by Art [4] in 1966, who used the term “shape envelope” to describe the positions where two polygons can be placed without intersection. This technique is used in many algorithms for (ISP) [1, 3, 9, 24, 25, 51], where Albano and Sapuppo [3] seems to be the first who used the term “no-fit polygon.” This concept is also used for other problems such as robot motion planning and image analysis, and is called in various names such as Minkowski sums and configuration-space obstacle. Practical algorithms to calculate the no-fit polygon of two non-convex polygons have been proposed, e.g., by Bennell et al. [10] and Ramkumar [53].

The no-fit polygon $\mathrm{NFP}(P_i, P_j)$ of an ordered pair of two polygons P_i and P_j is defined by

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

When the two polygons are clear from the context, we may simply use NFP instead of $\mathrm{NFP}(P_i, P_j)$. The no-fit polygon has the following important properties:

- $P_j \oplus \boldsymbol{v}_j$ overlaps with $P_i \oplus \boldsymbol{v}_i$ if and only if $\boldsymbol{v}_j - \boldsymbol{v}_i \in \mathrm{NFP}(P_i, P_j)$.
- $P_j \oplus \boldsymbol{v}_j$ touches $P_i \oplus \boldsymbol{v}_i$ if and only if $\boldsymbol{v}_j - \boldsymbol{v}_i \in \partial\mathrm{NFP}(P_i, P_j)$.
- $P_i \oplus \boldsymbol{v}_i$ and $P_j \oplus \boldsymbol{v}_j$ are separated if and only if $\boldsymbol{v}_j - \boldsymbol{v}_i \notin \mathrm{cl}(\mathrm{NFP}(P_i, P_j))$.

Hence the problem of checking whether two polygons overlap or not becomes an easier problem of checking whether a point is in a polygon or not. Figure 8 shows an example of $\mathrm{NFP}(P_i, P_j)$.

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

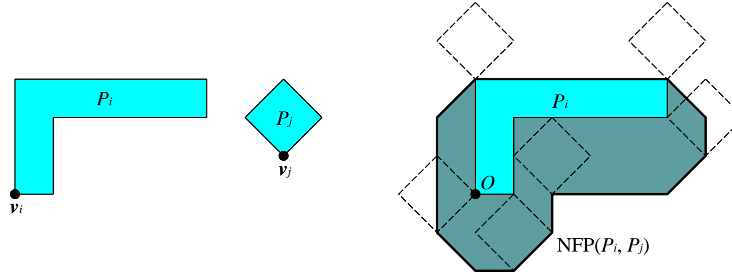


Fig. 8. An example of NFP(P_i, P_j), where O is the origin

We can also check whether a polygon P_i protrudes from the container C or not similarly by using NFP$(\bar{C}, P_i)$, which is the complement of a rectangle whose boundary is the trajectory of the reference point of P_i when we slide P_i inside C having it keep touching with C. (Gomes and Oliveira [24, 25] call $\overline{\mathrm{NFP}(\bar{C}, P_i)}$ *inner-fit rectangle*.)

5.3 Overlap minimization, compaction and separation

In this section, we introduce three important subproblems of (ISP), *overlap minimization*, *translational compaction* and *separation* problems [33, 40]. Algorithms for these problems will be discussed in the next section.

Overlap minimization problem For this problem, infeasible placements that have overlap and/or protrusion are allowed, and the height H of the container C is a given constant (e.g., temporarily fixed in a heuristic algorithm). For a given placement $\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n)$, let $f_{ij}(\boldsymbol{v})$ be a function that measures the amount of overlap of $P_i \oplus \boldsymbol{v}_i$ and $P_j \oplus \boldsymbol{v}_j$, and $g_i(\boldsymbol{v})$ be a function that measures the amount of protrusion of $P_i \oplus \boldsymbol{v}_i$ from the container $C(W^*, H)$. Then the objective of this problem is to find a placement $\boldsymbol{v} \in \mathbf{R}^{2n}$ that minimizes the total amount of overlap and protrusion

$$F(\boldsymbol{v}) = \sum_{1 \le i < j \le n} f_{ij}(\boldsymbol{v}) + \sum_{1 \le i \le n} g_i(\boldsymbol{v}).$$

It is not hard to see that this problem is NP-hard.

Translational compaction problem This problem is formulated as a two-dimensional motion planning problem. We are given a feasible placement $\boldsymbol{v}$ (i.e., $\boldsymbol{v}$ has no overlap or protrusion). The polygons and the container can move (translate) simultaneously, and the height H of the container can change. During a legal motion, the polygons cannot overlap each other nor protrude from the container. The objective is to minimize the height H of the container. See an example in Figure 9.

Li and Milenkovic [40] showed that this problem is PSPACE-hard. They also considered more general formulation, and mentioned different possibilities

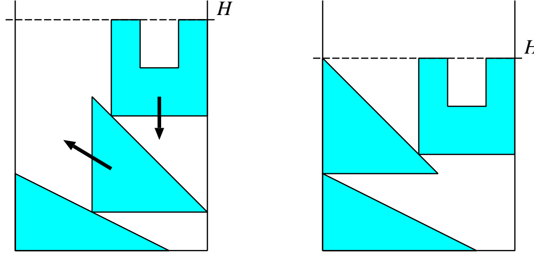


Fig. 9. An example of translational compaction

of utilizing this problem; e.g., to make a big hole in the given placement by moving polygons away from a given point.

Translational separation problem We are given an infeasible placement $\boldsymbol{v}$ (i.e., some polygons overlap and/or protrude from the container). The problem is to find a set of translations of the polygons that eliminates all overlaps and protrusion while minimizing the total amount of translation.

Li and Milenkovic [40] showed that simple special cases of this problem are NP-hard.

5.4 Nonlinear programming approach to overlap minimization

Imamichi et al. [33] considered the overlap minimization problem as an unconstrained nonlinear program, and incorporated a nonlinear programming technique in their heuristic algorithm. They defined the amount of overlap $f_{ij}(\boldsymbol{v})$ and protrusion $g_i(\boldsymbol{v})$ based on the concept of *penetration depth* so that they are smooth. Then they showed that differential coefficients $\nabla f_{ij}(\boldsymbol{v})$ and $\nabla g_i(\boldsymbol{v})$ can be computed efficiently, as they are needed in nonlinear programming algorithms.

The penetration depth (also known as the intersection depth) is an important notion used in robotics, computer vision and so on [2, 18, 37]. The penetration depth $\delta(P_i \oplus \boldsymbol{v}_i, P_j \oplus \boldsymbol{v}_j)$ of two overlapping polygons P_i and P_j placed at $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, respectively, is defined to be the minimum translational distance to separate them. If two polygons do not overlap, their penetration depth is zero. The formal definition of the penetration depth is given by

$$\delta(P_i \oplus \boldsymbol{v}_i, P_j \oplus \boldsymbol{v}_j) = \min\{||\boldsymbol{u}|| \mid \mathrm{int}(P_i \oplus \boldsymbol{v}_i) \cap (P_j \oplus \boldsymbol{v}_j \oplus \boldsymbol{u}) = \emptyset, \boldsymbol{u} \in \mathbf{R}^2\},$$

where $||\cdot||$ denotes the Euclidean norm.

The penetration depth $\delta(P_i \oplus \boldsymbol{v}_i, P_j \oplus \boldsymbol{v}_j)$ is the minimum distance from $\boldsymbol{v}_j - \boldsymbol{v}_i$ to the boundary $\partial\mathrm{NFP}(P_i, P_j)$ of the no-fit polygon. See an example in Figure 10, where arrow $\boldsymbol{u}$ is the minimum translation vector that separates the two polygons. Then the amounts of overlap and protrusion are defined by

$$\begin{aligned} f_{ij}(\boldsymbol{v}) &= \delta(P_i \oplus \boldsymbol{v}_i, P_j \oplus \boldsymbol{v}_j)^a, \quad 1 \leq i < j \leq n \\ g_i(\boldsymbol{v}) &= \delta(\mathrm{cl}(\bar{C}), P_i \oplus \boldsymbol{v}_i)^a, \qquad 1 \leq i \leq n, \end{aligned}$$

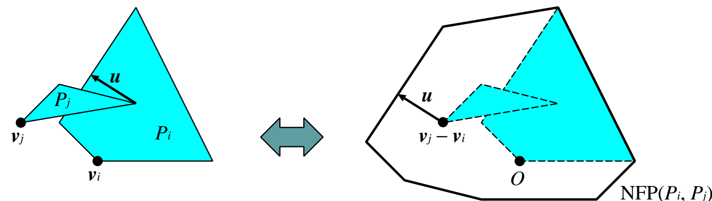


Fig. 10. The computation of penetration depth via no-fit polygon

where $a > 0$ is a parameter. Under this definition, for any placement $\boldsymbol{v}$, the function values $f_{ij}(\boldsymbol{v})$ and $g_i(\boldsymbol{v})$ as well as $\nabla f_{ij}(\boldsymbol{v})$ and $\nabla g_i(\boldsymbol{v})$ can be computed efficiently by using no-fit polygons. They set $a = 2$ for the following two reasons:

- At the boundary of no-fit polygons (i.e., when two polygons touch each other), $f_{ij}(\boldsymbol{v})$ and $g_i(\boldsymbol{v})$ are differentiable if and only if $a > 1$.
- The formulae of $\nabla f_{ij}(\boldsymbol{v})$ and $\nabla g_i(\boldsymbol{v})$ become the simplest when $a = 2$.

Then the problem becomes an unconstrained quadratic programming problem (e.g., [11]), for which many efficient algorithms for finding locally optimal solutions exist; e.g., quasi-Newton method, conjugate gradient method, etc.

5.5 Linear programming approach to translational compaction and separation

Li and Milenkovic [40] proposed linear programming (LP) approaches for translational compaction and separation problems, which are called the compaction and separation algorithms, respectively. Similar ideas are also used in [9, 25, 54]. We first explain their method for compaction.

The main idea is to restrict the search to a convex subregion of the original problem, which is realized by adding artificial linear constraints, in order to apply linear programming methods. They call the heuristic rules to add such constraints *locality heuristics*. The subregion should contain the given placement, and a larger region is preferable.

Let $\boldsymbol{v}^{(0)}$ be the given placement and $\Delta\boldsymbol{v}$ be the translation added to $\boldsymbol{v}^{(0)}$ (i.e., $\boldsymbol{v}^{(0)}$ is the given constant, $\Delta\boldsymbol{v}$ is the decision variable, and $\boldsymbol{v}^{(0)} + \Delta\boldsymbol{v}$ gives the modified placement). Among the constraints of problem (ISP), only the first one

$$\mathrm{int}(P_i \oplus (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i)) \cap (P_j \oplus (\boldsymbol{v}_j^{(0)} + \Delta\boldsymbol{v}_j)) = \emptyset, \quad 1 \le i < j \le n,$$

which is equivalent to

$$(\boldsymbol{v}_j^{(0)} + \Delta\boldsymbol{v}_j) - (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i) \notin \mathrm{NFP}(P_i, P_j), \quad 1 \le i < j \le n,$$

is nonconvex, and others are convex linear. The objective of the locality heuristics is to define, for each pair of P_i and P_j, a subset $S_{ij}(\boldsymbol{v}^{(0)}) \subseteq \overline{\mathrm{NFP}(P_i, P_j)}$ that has the following properties: (1) convex, (2) large, and (3) contains $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$. Then, for such subsets $S_{ij}(\boldsymbol{v}^{(0)})$, it is not hard to see that the following problem is a linear programming problem:

$$\begin{array}{lll}
\text{minimize} & H & \\
\text{subject to} & (\boldsymbol{v}_j^{(0)} + \Delta\boldsymbol{v}_j) - (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i) \in S_{ij}(\boldsymbol{v}^{(0)}), & 1 \le i < j \le n \\
& (P_i \oplus (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i)) \subseteq C(W^*, H), & 1 \le i \le n \\
& H \ge 0, & \\
& \Delta\boldsymbol{v}_i \in \mathbf{R}^2, & 1 \le i \le n.
\end{array}$$

Note that Li and Milenkovic [40] used a different objective function $\sum_{P_i \in \mathcal{P}} \boldsymbol{d}_i \boldsymbol{v}_i$ in their LP formulation, where $\boldsymbol{d}_i$ is a desirable direction to move each polygon P_i, and the above formulation is due to [9, 25].

Figure 11 is an example of a subregion outside a no-fit polygon. The left figure shows the given placement $\boldsymbol{v}_i^{(0)}$ and $\boldsymbol{v}_j^{(0)}$ of two polygons P_i and P_j, while the right figure shows the corresponding position of $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$ against no-fit polygon $\mathrm{NFP}(P_i, P_j)$, and a convex set $S_{ij}(\boldsymbol{v}^{(0)}) \subseteq \overline{\mathrm{NFP}(P_i, P_j)}$ that satisfies $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)} \in S_{ij}(\boldsymbol{v}^{(0)})$.

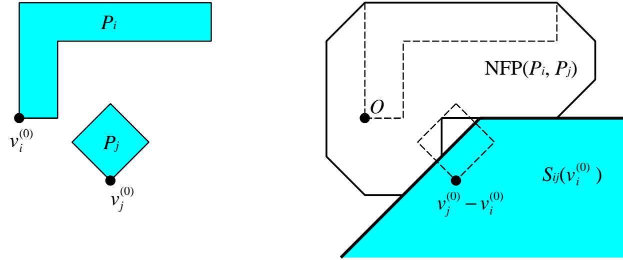


Fig. 11. A convex subregion $S_{ij}(\boldsymbol{v}^{(0)})$ outside $\mathrm{NFP}(P_i, P_j)$

The locality heuristic procedure in [40] defines the boundary of $S_{ij}(\boldsymbol{v}^{(0)})$ as follows. Starting from an edge of $\mathrm{NFP}(P_i, P_j)$ close to[7] $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$, it walks along $\partial\mathrm{NFP}(P_i, P_j)$ in both clockwise and counterclockwise directions, and the trace of the walk becomes the boundary $\partial S_{ij}(\boldsymbol{v}^{(0)})$. When walking clockwise, $\partial S_{ij}(\boldsymbol{v}^{(0)})$ should make only left turns to keep $S_{ij}(\boldsymbol{v}^{(0)})$ convex. If the next edge of NFP turns to the left (i.e., a concave vertex of NFP), the walk follows

[7] Li and Milenkovic [40] consider a special type of polygons called star-shaped, and define the origins of no-fit polygons accordingly. Then the edge that crosses with the line segment from the origin to the point $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$ is chosen. This is not necessarily the closest edge.

it; otherwise, the walk extends the current edge until it intersects with NFP. This procedure continues until it can extend the current edge infinitely. The walk to the counterclockwise direction is similar. The resulting $S_{ij}(\boldsymbol{v}^{(0)})$ can be different if the starting edge is different. Gomes and Oliveira [25] suggest to choose a convex subregion whose closest edge from $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$ is most distant. Intuitively, this rule has an effect of making more margin for compaction.

Given an optimal solution $\Delta\boldsymbol{v}^*$ to the above LP problem, we can repeat the same procedure from the new placement $\boldsymbol{v}^{(1)} = \boldsymbol{v}^{(0)} + \Delta\boldsymbol{v}^*$. Hence we can generate a sequence of improved placements $\boldsymbol{v}^{(0)}, \boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \ldots$ until the objective values of $\boldsymbol{v}^{(k)}$ and $\boldsymbol{v}^{(k+1)}$ become the same for some k. It is not hard to observe that any convex combination of two consecutive placements $t\boldsymbol{v}^{(l)} + (1-t)\boldsymbol{v}^{(l+1)}$ $(0 \leq t \leq 1,\ l = 0, 1, \ldots, k)$ is feasible. Hence such a sequence of placements gives a (piecewise linear) legal motion to the translational compaction problem.

Here it should be noted that having the constraints $(\boldsymbol{v}_j^{(0)} + \Delta\boldsymbol{v}_j) - (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i) \in S_{ij}(\boldsymbol{v}^{(0)})$ for all pairs of polygons is not necessary in practice, and is time consuming. Hence such constraints are usually imposed only for relatively close pairs in the current placement [9, 40].

Similar technique is applicable to the translational separation problem. For a given infeasible placement $\boldsymbol{v}^{(0)}$, we can define a convex subregion $S_{ij}(\boldsymbol{v}^{(0)})$ similarly even for overlapping polygons; e.g., by making a walk starting from an edge close to $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)}$, where the constraint $\boldsymbol{v}_j^{(0)} - \boldsymbol{v}_i^{(0)} \in S_{ij}(\boldsymbol{v}^{(0)})$ cannot hold in this case. Then the objective of the resulting LP problem, whose constraints are the same as the LP model for compaction, is to find a feasible placement that is close to $\boldsymbol{v}^{(0)}$.

If the above LP problem is infeasible, then no feasible placement is found. To deal with such situations, Bennell and Dowsland [9] relax the violated constraints slightly and solve the modified LP problem again; then repeat a limited number of such steps. Gomes and Oliveira [25] consider a modified formulation in which the objective is the sum of the penalty for violating constraints $(\boldsymbol{v}_j^{(0)} + \Delta\boldsymbol{v}_j) - (\boldsymbol{v}_i^{(0)} + \Delta\boldsymbol{v}_i) \in S_{ij}(\boldsymbol{v}^{(0)})$. This can be regarded as an algorithm for a variant of the overlap minimization problem.

5.6 Hybrid approaches

In this section, we summarize hybrid metaheuristic approaches for the irregular strip packing problem.

Imamichi et al. [33] proposed an iterated local search (ILS) algorithm, in which the nonlinear programming technique in Section 5.4 is incorporated. (The framework of ILS was explained in Section 2.3.) The core part of their ILS is the algorithm for the overlap minimization problem. They fix the height H of the container temporarily, and solve the overlap minimization problem. If a feasible placement is found (resp., not found), they reduce (resp., increase) H slightly, and solve the overlap minimization problem again. Such

iterations are repeated until some stopping criterion is met. They used the quasi-Newton method for the nonlinear programming formulation of the overlap minimization problem, explained in Section 5.4. Given an initial solution (to the overlap minimization problem), the quasi-Newton method is applied, which can be viewed as local search since it iteratively improves the current solution by applying slight modifications to it until a locally optimal solution is found. This local search is iterated from different initial solutions, and the entire algorithm is regarded as ILS. The perturbation for generating the next initial solution is realized by a swap of the positions of two polygons, which are found by a sophisticated algorithm based on no-fit polygons under the condition that other polygons do not move.

There are other metaheuristic algorithms based on different formulations of the overlap minimization problem. Umetani et al. [57] defines the penalty of two overlapping polygons to be the minimum translational distance in a specified direction (e.g., horizontal or vertical direction) to separate them. Egeblad et al. [21] defines the penalty of two overlapping polygons to be their overlapping area. In these papers, they devise efficient algorithms to find the best position of a polygon when it is translated to a specified direction, and use them to define neighborhood operations. Such efficient neighborhood search algorithms are then incorporated in guided local search framework in both papers. They can be regarded as hybridization of metaheuristics and algorithmic techniques for computational geometry.

Compaction approaches can modify given feasible placements to more efficient ones, but cannot perform wider search. This limitation is shown in [7], which finds that the compaction of randomly generated solutions cannot compete with local search. Thus, it seems meaningful to combine metaheuristics with compaction/separation algorithms.

Bennell and Dowsland [9] proposed a hybrid approach of separation/compaction and tabu thresholding algorithm, whose original version without hybridization was proposed in [8]. Their algorithm basically deals with the overlap minimization problem whose objective function is similar to [57]. Their neighborhood operation is to move a polygon to another position in the container. Tabu thresholding, proposed in [22], is a variant of tabu search. It consists of two phases called the improving phase and the mixed phase, and these phases are alternately repeated. The improving phase is a standard local search, while non-improving moves are allowed in the mixed phase. In the mixed phase, the neighborhood is divided into subareas, and for each move one of them is chosen and the best neighbor in it is chosen even if it is worse than the current solution. They apply separation and compaction for the locally optimal solutions obtained after improving phases, but they limit the application of separation/compaction only for promising solutions because it is computationally expensive.

Gomes and Oliveira [25] proposed a hybrid approach of separation/compaction with simulated annealing. They limit the search space to feasible placements, and allow infeasible placements only when trial solutions are gen-

erated by the neighborhood operation, where they adopt the swap neighborhood (i.e., the positions of the two polygons are exchanged). Whenever a trial solution is generated, separation and compaction algorithms are applied, and the new placement is evaluated by the height H of the container if it is feasible, and is discarded if separation fails.

5.7 Computational results

We briefly report some computational results of algorithms which give high quality solutions. They are (1) the iterated local search incorporated with quasi-Newton method (denoted as ILSQN) proposed in [33], (2) the simulated annealing incorporated with separation and compaction (denoted as SAHA) proposed in [25], and (3) the guided local search (denoted as 2DNest) proposed in [21]. The instances are available from the ESICUP web site.[8] For these instances, rotations of fixed degrees are allowed.

Table 5 shows their efficiency in % and computation time in seconds, where the efficiency of a solution is measured by the ratio $\sum_{P_i \in \mathcal{P}}(\text{area of } P_i)/W^*H$, which is shown in column "EF." The results were taken from their original papers unless otherwise stated. (The results of ILSQN are those reported in the full paper version of [33], which was submitted for publication.) Note that the stopping criteria of these algorithms are different; column "time" shows the time limit of each run for ILSQN and 2DNest, while it shows the average computation time of all runs for SAHA. Column NDP, TNP and ANV show the number of different polygons, the total number of polygons, and the average number of vertices of different polygons, respectively. The value with "†" has been corrected from the one reported in [25] according to the information sent from the authors. The best results among the three algorithms are marked with "$*$." From the table, we can observe that ILSQN is somewhat better than the others.

References

1. Adamowicz M, Albano A (1976) Nesting two-dimensional shapes in rectangular modules, Computer-Aided Design 8:27-33
2. Agarwal PK, Guibas LJ, Har-Peled S, Rabinovitch A, Sharir M (2000) Penetration depth of two convex polytopes in 3D, Nordic Journal of Computing 7:227–240
3. Albano A, Sapuppo G (1980) Optimal allocation of two-dimensional irregular shapes using heuristic search methods, IEEE Transactions on Systems, Man and Cybernetics 10:242–248
4. Art Jr. RC (1966) An approach to the two-dimensional irregular cutting stock problem, Technical Report 36-Y08, IBM Cambridge Science Center

[8] http://www.fe.up.pt/esicup/

Table 5. Comparison of three algorithms for the irregular strip packing problem

instance	NDP	TNP	ANV	ILSQN		SAHA		2DNest	
				EF(%)	time(s)	EF(%)	time(s)	EF(%)	time(s)
ALBANO	8	24	7.25	*88.16	1200	87.43	2257	87.44	600
DAGLI	10	30	6.30	*87.40	1200	87.15	5110	85.98	600
DIGHE1	16	16	3.87	99.89	600	*100.00	83	99.86	600
DIGHE2	10	10	4.70	99.99	600	*100.00	22	99.95	600
FU	12	12	3.58	90.67	600	90.96	296	*91.84	600
JAKOBS1	25	25	5.60	86.89	600	†78.89	332	*89.07	600
JAKOBS2	25	25	5.36	*82.51	600	77.28	454	80.41	600
MAO	9	20	9.22	83.44	1200	82.54	8245	*85.15	600
MARQUES	8	24	7.37	89.03	1200	88.14	7507	*89.17	600
SHAPES0	4	43	8.75	*68.44	1200	66.50	3914	67.09	600
SHAPES1	4	43	8.75	*73.84	1200	71.25	10314	*73.84	600
SHAPES2	7	28	6.29	*84.25	1200	83.60	2136	81.21	600
SHIRTS	8	99	6.63	*88.78	1200	86.79	10391	86.33	600
SWIM	10	48	21.90	*75.29	1200	74.37	6937	71.53	600
TROUSERS	17	64	5.06	89.79	1200	*89.96	8588	89.84	600
CPU					Xeon		Pentium4		Pentium4
clock freq.					2.8 GHz		2.4 GHz		3.0 GHz
#runs					10		20		20

5. Babu AR, Babu NR (2001) A genetic approach for nesting of 2-D parts in 2-D sheets using genetic and heuristic algorithms, Computer-Aided Design 33:879–891
6. Baker BS, Coffman Jr. EG, Rivest RL (1980) Orthogonal packing in two dimensions, SIAM Journal on Computing 9:846–855
7. Bennell JA (1998) Incorporating problem specific knowledge into a local search framework for the irregular shape packing problem, Ph.D. thesis, European Business Management School, University of Wales, Swansea
8. Bennell JA, Dowsland KA (1999) A tabu thresholding implementation for the irregular stock cutting problem, International Journal of Production Research 37:4259–4275
9. Bennell JA, Dowsland KA (2001) Hybridising tabu search with optimisation techniques for irregular stock cutting, Management Science 47:1160–1172
10. Bennell JA, Dowsland KA, Dowsland WB (2001) The irregular cutting-stock problem—a new procedure for deriving the no-fit polygon, Computers & Operations Research 28:271–287
11. Bertsekas DP (1999) Nonlinear Programming (2nd Edition), Athena Scientific.
12. Błażewicz J, Hawryluk P, Walkowiak R (1993) Using a tabu search for solving the two-dimensional irregular cutting problem, Annals of Operations Research 41:313–325
13. Burke E, Hellier R, Kendall G, Whitwell G (2006) A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem, Operations Research 54:587–601
14. Burke EK, Kendall G, Whitwell G (2004) A new placement heuristic for the orthogonal stock-cutting problem, Operations Research 52: 655–671

15. Chang YC, Chang YW, Wu GM, Wu SW (2000) B*-trees: a new representation for non-slicing floorplans, In: Proceedings of the 37th Design Automation Conference, 458–463
16. Chu CCN, Young EFY (2004) Nonrectangular shaping and sizing of soft modules for floorplan-design improvement, IEEE Transactions Computer Aided Design of Integrated Circuits and Systems 23:71–79
17. Coffman Jr. EG, Garey MR, Johnson DS, Tarjan RE (1980) Performance bounds for level-oriented two-dimensional packing algorithms, SIAM Journal on Computing 9:801–826
18. Dobkin D, Hershberger J, Kirkpatrick D, Suri S (1993) Computing the intersection-depth of polyhedra, Algorithmica 9:518–533
19. Dréo J, Pétrowski JDA, Siarry P, Taillard E (2006) Metaheuristics for Hard Optimization, Springer.
20. Dyckhoff H (1990) A typology of cutting and packing problems, European Journal of Operational Research 44:145–159
21. Egeblad J, Nielsen BK, Odgaard A (to appear) Fast neighborhood search for two- and three-dimensional nesting problems, European Journal of Operational Research
22. Glover F (1992) Simple tabu thresholding in optimization, Internal report, University of Colorado, Boulder, CO
23. Glover FW, Kochenberge GA (eds) (2003) Handbook of Metaheuristics, Springer.
24. Gomes AM, Oliveira JF (2002) A 2-exchange heuristic for nesting problems, European Journal of Operational Research 141:359–370
25. Gomes AM, Oliveira JF (2006) Solving irregular strip packing problems by hybridising simulated annealing and linear programming, European Journal of Operational Research 171:811–829
26. Guo PN, Takahashi T, Cheng CK, Yoshimura T (2001) Floorplanning using a tree representation, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 20:281–289
27. Heckmann R, Lengauer T (1995) A simulated annealing approach to the nesting problem in the textile manufacturing industry, Annals of Operations Research 57:103–133
28. Hopper E, Turton BCH (2001) An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem, European Journal of Operational Research 128:34–57
29. Ibaraki T, Nakamura K (2006) Packing problems with soft rectangles, In: Almeida F, Aguilera MJB, Blum C, Vega JMM, Pérez MP, Roli A, Sampels M (eds) Hybrid Metaheuristics, Springer Lecture Notes on Computer Science 4030:13–27
30. Imahori S, Yagiura M, Ibaraki T (2003) Local search algorithms for the rectangle packing problem with general spatial costs, Mathematical Programming 97:543–569
31. Imahori S, Yagiura M, Ibaraki T (2005) Improved local search algorithms for the rectangle packing problem with general spatial costs, European Journal of Operational Research 167:48–67
32. Imahori S, Yagiura M, Ibaraki T (2005) Variable neighborhood search for the rectangle packing problem, In: Proceedings of the 6th Metaheuristics International Conference

33. Imamichi T, Yagiura M, Nagamochi H (2006) An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem, In: Proceedings of the Third International Symposium on Scheduling (ISS06) 132–137
34. Itoga H, Kodama C, Fujiyoshi K (2005) A graph based soft module handling in floorplan, IEICE Transactions Fundamentals E88-A:3390–3397
35. Johnson DS (1990) Local optimization and the traveling salesman problem, In: Peterson MS (ed) Automata, Languages and Programming, Lecture Notes in Computer Science 443:446–461
36. Kenyon C, Rémila E (2000) A near-optimal solution to a two-dimensional cutting stock problem, Mathematics of Operations Research 25:645–656
37. Kim YJ, Lin MC, Manocha D (2004) Incremental penetration depth estimation between convex polytopes using dual-space expansion, IEEE Transactions on Visualization and Computer Graphics 10:152–163
38. Konno H, Kuno T (1995) Multiplicative programming problems, In: Horst R, Pardalos PM (eds) Handbook of Global Optimization, Kluwer Academic Publishers, 369–406
39. Kurebe Y, Miwa H, Ibaraki T (2006) Weighted module placement based on rectangle packing, submitted to an international conference.
40. Li Z, Milenkovic V (1995) Compaction and separation algorithms for nonconvex polygons and their applications, European Journal of Operational Research 84:539–561
41. Lodi A, Martello S, Monaci M (2002) Two-dimensional packing problems: A survey, European Journal of Operational Research 141:241–252
42. Lodi A, Martello S, Vigo D (1999) Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems, INFORMS Journal on Computing 11:345–357
43. Milenkovic VJ (1998) Rotational polygon overlap minimization and compaction, Computational Geometry 10:305–318
44. Murata H, Fujiyoshi K, Nakatake S, Kajitani Y (1996) VLSI module placement based on rectangle-packing by the sequence-pair, IEEE Transactions on Computer Aided Design 15:1518–1524
45. Murata H, Kuh ES (1998) Sequence-pair based placement method for hard/soft/preplaced modules, In: Proceedings of International Symposium on Physical Design, 167–172
46. Nagamochi H (2005) Packing soft rectangles, International Journal of Foundations of Computer Science 17:1165–1178
47. Nakatake S, Fujiyoshi K, Murata H, Kajitani Y (1998) Module packing based on the BSG-structure and IC layout applications, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 17:519–530
48. Nesterov Y, Nemirovskii A (1994) Interior Point Polynomial Algorithms in Convex Programming, SIAM Pub.
49. Okano H (2002) A scanline-based algorithm for the 2D free-form bin packing problem, Journal of the Operations Research Society of Japan 45:145–161
50. Oliveira JF, Ferreira JS (1993) Algorithms for nesting problems, In: Vidal RVV (ed) Applied Simulated Annealing. Lecture Notes in Economics and Mathematical Systems 396, Springer-Verlag, 255–274
51. Oliveira JF, Gomes AM, Ferreira JS (2000) TOPOS—a new constructive algorithm for nesting problems, OR Spektrum 22:263–284

52. Preas BT, van Cleemput WM (1979) Placement algorithms for arbitrarily shaped blocks, In: Proceedings of the ACM/IEEE Design Automation Conference, 474–480
53. Ramkumar GD (1996) An algorithm to compute the Minkowski sum outer-face of two simple polygons, In: Proceedings of the Twelfth Annual Symposium on Computational Geometry (SCG96), 234–241
54. Stoyan YG, Novozhilova MV, Kartashov AV (1996) Mathematical model and method of searching for a local extremum for the non-convex oriented polygons allocation problem, European Journal of Operational Research 92:193–210
55. Takahashi T (1996) An algorithm for finding a maximum-weight decreasing sequence in a permutation, motivated by rectangle packing problem (in Japanese), Technical Report of IEICE VLD96-30, 31–35
56. Tang X, Tian R, Wong DF (2001) Fast evaluation of sequence pair in block placement by longest common subsequence computation, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 20:1406–1413
57. Umetani S, Yagiura M, Imamichi T, Imahori S, Nonobe K, Ibaraki T (2006) A guided local search algorithm based on a fast neighborhood search for the irregular strip packing problem, In: Proceedings of the Third International Symposium on Scheduling (ISS06), 126–131
58. Wäscher G, Haußner H, Schumann H (2004) An improved typology of cutting and packing problems, Working Paper 24, Faculty of Economics and Management, Guericke University Magdeburg
59. Young FY, Chu CCN, Luk WL, Wong YC (2001) Handling soft modules in general nonslicing floorplan using Lagrangean relaxation, IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systetems 20: 687–692