Mathematical Programming manuscript No.
(will be inserted by the editor)

S. Imahori · M. Yagiura · T. Ibaraki

Local Search Algorithms for the Rectangle Packing Problem with General Spatial Costs

Received: 15 April 2002 / Revised version: 27 September 2002

Abstract. We propose local search algorithms for the rectangle packing problem to minimize a general spatial cost associated with the locations of rectangles. The problem is to pack given rectangles without overlap in the plane so that the maximum cost of the rectangles is minimized. Each rectangle has a set of modes, where each mode specifies the width and height of the rectangle and its spatial cost function. The spatial costs are general piecewise linear functions which can be non-convex and discontinuous. Various types of packing problems and scheduling problems can be formulated in this form. To represent a solution of this problem, a pair of permutations of n rectangles is used to determine their horizontal and vertical partial orders, respectively. We show that, under the constraint specified by such a pair of permutations, optimal locations of the rectangles can be efficiently determined by using dynamic programming. The search for finding good pairs of permutations is conducted by local search and metaheuristic algorithms. We report computational results on various implementations using different neighborhoods, and compare their performance. We also compare our algorithms with other existing heuristic algorithms for the rectangle packing problem and scheduling problem. These computational results exhibit good prospects of the proposed algorithms.

Keywords: rectangle packing – sequence pair – general spatial cost – dynamic programming – metaheuristics.

1. Introduction

The two dimensional rectangle packing problem (2RP) is to place a set of rectangles in the plane without overlap so that a given cost function is minimized. In this paper, the edges of rectangles are required to be vertical or horizontal. There are various options on the packing rules and objective functions; e.g., rotations are allowed or not, a bin of fixed size to which rectangles are packed is given or not. The problem is known to be NP-hard in general, and hence a number of heuristic algorithms have been proposed in the literature [9–12].

Liu and Teng [9] proposed an algorithm for 2RP, based on the genetic algorithm in which rotations of 90° are allowed and the width W of the rectangle bin is given. The objective is to minimize the height of the bin containing all given rectangles. Lodi, Martello and Vigo [10] proposed simple heuristic algorithms and incorporated them into a metaheuristic algorithm based on the tabu search. They solved 2RP in which the size of the rectangle bins to which given rectangles should be packed is given. The objective is to minimize the number of bins.

Authors: Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan, Phone/Fax: +81-75-753-5494,
E-mail: {imahori,yagiura,ibaraki}@amp.i.kyoto-u.ac.jp

Mathematics Subject Classification (1991): 20E28, 20G40, 20C20

Murata, Fujiyoshi, Nakatake and Kajitani [11] proposed a simulated annealing algorithm for 2RP in which rotations of 90° are allowed and the objective is to minimize the area of the bin containing all given rectangles.

In this paper, we consider the rectangle packing problem with general spatial costs (RPGSC), and propose metaheuristic algorithms based on local search. Each rectangle is associated with a set of modes, where each mode specifies the width and height of the rectangle and its spatial cost function. The spatial costs are general piecewise linear functions which can be non-convex and discontinuous. The problem is to pack a given set of n rectangles without overlap so that the maximum cost of the rectangles is minimized. A solution, called a packing, is determined by specifying the mode and the location of each rectangle, which is feasible if no two rectangles overlap.

This problem is very general, and various types of packing problems and scheduling problems can be formulated in this form. For example, we can treat 2RP in which rotations of 90° are allowed as a special case of RPGSC by considering two modes for each rectangle corresponding to: (1) the original orientation and (2) the orientation rotated by 90°. A quite different problem taken from the scheduling problem of constructing large building blocks can also be handled by using appropriate spatial costs, as will be described in Section 5.

If we search directly the x and y coordinates and the mode of each rectangle, then an effective search will be difficult, since the number of solutions is uncountably many and eliminating overlap of rectangles is not easy. Therefore various coding schemes to represent solutions have been proposed [9, 11, 12]. Nakatake, Fujiyoshi, Murata and Kajitani proposed a coding scheme called the bounded-sliceline grid (BSG) [12], and obtained good results for 2RP in which rotations are allowed and the objective is to minimize the area of the bin containing all given rectangles. Murata, Fujiyoshi, Nakatake and Kajitani proposed a coding scheme called the sequence pair [11]. In the sequence pair representation, a solution is represented by a pair of permutations of rectangles. They proposed an $O(n^2)$ time decoding algorithm to obtain a feasible packing (i.e., x and y coordinates of rectangles) from a pair of permutations of rectangles. Takahashi [15] and Tang, Tian and Wong [16] improved the time complexity of the decoding algorithm to $O(n \log n)$; Tang and Wong [17] further improved it to $O(n \log \log n)$.

We adopt the sequence pair [11] as the coding scheme in our algorithm. A solution is coded as a pair of permutations of n rectangles and a vector specifying the modes of all rectangles. Given a coded solution, we propose a decoding algorithm based on dynamic programming to obtain an optimal packing (i.e., locations of the rectangles that minimize the associated cost function) under the constraint specified by the coded solution. This algorithm is a generalization of the algorithms proposed in [15, 16] in that it can deal with general spatial costs. We propose another decoding algorithm which slightly relaxes the constraints of a sequence pair and finds a packing better than or equivalent to that obtained by our first algorithm. The running time of these algorithms is $O(n \log n)$ if applied to the case of area minimization. The details of these algorithms are described in Section 3.

We also propose an encoding algorithm to obtain a coded solution from a given packing, which runs in $O(n \log n)$ time. This encoding algorithm is incorporated in our local search and metaheuristic algorithms whenever our second decoding algorithm is used. The details of this algorithm are described in Appendix A.

We then consider various neighborhoods in Section 4, which are used in the local search algorithms for finding good coded solutions. We define the critical path (its definition is not trivial as we consider general spatial cost functions), which represents the bottleneck of the current solution, and propose a neighborhood based on critical paths. We also use critical paths to reduce the sizes of other neighborhoods. The local search algorithms based on these neighborhoods are then incorporated in metaheuristic algorithms such as the multi-start local search (MLS) and the iterated local search (ILS).

The computational results are reported in Section 5. We compare the performance of various implementations using different neighborhoods and metaheuristics. We also compare our algorithms with other existing heuristic algorithms for the rectangle packing problem and a real-world scheduling problem.

2. Rectangle packing problem with general spatial costs

In this section, we formulate the rectangle packing problem with general spatial costs. Let $I = \{1, 2, \ldots, n\}$ be a set of n rectangles. Each rectangle $i \in I$ has m_i modes, and each mode k ($k = 1, 2, \ldots, m_i$) of rectangle i specifies:

- a width $w_i^{(k)}$, a height $h_i^{(k)}$ ($w_i^{(k)}, h_i^{(k)} \geq 0$) and a cost $c_i^{(k)}$ of the mode,
- spatial cost functions $p_i^{(k)}(x)$ and $q_i^{(k)}(y)$ on the location (x, y) of the rectangle, where the location of a rectangle means the (x, y)-coordinate of its lower left corner.

We assume that $p_i^{(k)}(x)$ and $q_i^{(k)}(y)$ are piecewise linear and nonnegative (i.e., $p_i^{(k)}(x), q_i^{(k)}(y) \geq 0$ hold for all $x, y \in [-\infty, \infty]$). It is also assumed that if $x, y \to \pm\infty$, then $p_i^{(k)}(x), q_i^{(k)}(y) \to +\infty$. Moreover, we assume that any discontinuous point x (if exists) must satisfy

$$p_i^{(k)}(x) \leq \min\{\lim_{s \to +\infty} p_i^{(k)}(x + 1/s), \lim_{s \to +\infty} p_i^{(k)}(x - 1/s)\}$$

(see Fig. 1 as an example). The assumption on the discontinuous points of $q_i^{(k)}(y)$ is similar. The latter two conditions are necessary to ensure the existence of an optimal solution, and most of natural spatial cost functions satisfy them. Note that the spatial cost functions can be non-convex and discontinuous as long as they satisfy the above conditions. It is also assumed throughout the paper (unless otherwise stated) that the information of each linear piece of functions $p_i^{(k)}(x)$ and $q_i^{(k)}(y)$ are given explicitly. In many applications, the number of linear pieces of each spatial cost function is small, and hence this assumption is natural. Given the modes of n rectangles

$$\mu(\pi) = (\mu_1(\pi), \mu_2(\pi), \ldots, \mu_n(\pi)),$$

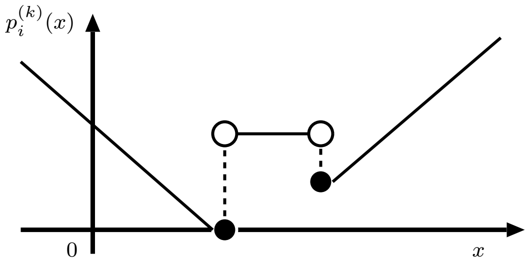


Fig. 1. An example of the spatial cost function

a packing π is determined by locations $(x_i(\pi), y_i(\pi))$ of all rectangles i. We define $p_{\max}(\pi)$ and $q_{\max}(\pi)$ as follows:

$$p_{\max}(\pi) = \max_{i \in I} p_i^{(\mu_i(\pi))}(x_i(\pi)), \tag{1}$$

$$q_{\max}(\pi) = \max_{i \in I} q_i^{(\mu_i(\pi))}(y_i(\pi)). \tag{2}$$

Now we are given two cost functions $g(p_{\max}(\pi), q_{\max}(\pi))$ and $c(\mu(\pi))$, where function g is nondecreasing in $p_{\max}(\pi)$ and $q_{\max}(\pi)$. Moreover, we assume that $g(p_{\max}(\pi), q_{\max}(\pi))$ (respectively, $c(\mu(\pi))$) can be computed in $O(1)$ (respectively, $O(n)$) time for given $p_{\max}(\pi)$ and $q_{\max}(\pi)$ (respectively, $\mu(\pi)$). Then the problem considered in this paper is defined as follows:

RPGSC: minimize $g(p_{\max}(\pi), q_{\max}(\pi)) + c(\mu(\pi))$

subject to At least one of the next four inequalities holds for every pair (i, j) of rectangles:

$$x_i(\pi) + w_i^{(\mu_i(\pi))} \le x_j(\pi), \tag{3}$$

$$x_j(\pi) + w_j^{(\mu_j(\pi))} \le x_i(\pi), \tag{4}$$

$$y_i(\pi) + h_i^{(\mu_i(\pi))} \le y_j(\pi), \tag{5}$$

$$y_j(\pi) + h_j^{(\mu_j(\pi))} \le y_i(\pi). \tag{6}$$

The constraints from (3) to (6) mean that no two rectangles overlap in packing π, and we call a packing satisfying all constraints feasible. For example, condition (3) means that the right side of rectangle i is placed to the left of the left side of rectangle j.

Typical examples of $g(p_{\max}(\pi), q_{\max}(\pi))$ are

$$g(p_{\max}(\pi), q_{\max}(\pi)) = p_{\max}(\pi) + q_{\max}(\pi),$$
$$g(p_{\max}(\pi), q_{\max}(\pi)) = p_{\max}(\pi) \cdot q_{\max}(\pi),$$
$$g(p_{\max}(\pi), q_{\max}(\pi)) = \max\{p_{\max}(\pi), q_{\max}(\pi)\},$$

and typical examples of $c(\mu(\pi))$ are

$$c(\mu(\pi)) = \sum_{i \in I} c_i^{(\mu_i(\pi))},$$

$$c(\mu(\pi)) = \max_{i \in I} c_i^{(\mu_i(\pi))}.$$

Defining $p_i^{(k)}(x)$, $q_i^{(k)}(y)$ and $c_i^{(k)}$ for each i and k appropriately, various types of packing problems and scheduling problems can be formulated in our form. For example, we can treat 2RP in which rotations of 90° are allowed as a special case of RPGSC by considering two modes of (1) the original orientation and (2) the orientation rotated by 90°. We can also treat both types of rectangle packing problems, where a bin of fixed size to which rectangles should be packed is given or not, by defining $p_i^{(k)}(x)$ and $q_i^{(k)}(y)$ appropriately. We will explain in Section 5.1 that some other problems taken from scheduling applications can be formulated as RPGSC. Some computational results of our algorithms and other existing algorithms are reported in Section 5.

The problem RPGSC is NP-hard, since the one dimensional bin packing problem, which is known to be strongly NP-hard [5], can be polynomially reducible to this problem. Note that problem RPGSC is not purely combinatorial, since the coordinates $x_i(\pi)$ and $y_i(\pi)$ of each rectangle i can be any real values and hence there are continuously infinite number of solutions.

3. Packing and sequence pair

As noted before, various coding schemes to represent solutions have been proposed [9, 11, 12]. Desirable properties of a coding scheme may be summarized as follows.

1. The size of the search space (i.e., the number of all possible coded solutions) is finite.
2. Every coded solution corresponds to a feasible packing.
3. Decoding (i.e., computing the corresponding packing from a coded solution) is possible in polynomial time.
4. There exists a coded solution that corresponds to an optimal packing.

Some of the coding schemes in the literature satisfy all of the above four properties, and others do not. We adopt the sequence pair [11] as the coding scheme in our algorithm, which satisfies the above four properties. In this scheme, a coded solution is a pair of permutations of n rectangles and a vector that specifies the modes of all rectangles. In this section, we briefly explain the sequence pair representation and propose decoding algorithms to find a feasible packing from a coded solution.

3.1. Sequence pair

A sequence pair is a pair of permutations $\sigma = (\sigma_+, \sigma_-)$ of $I = \{1, 2, \ldots, n\}$, where $\sigma_+(l) = i$ (equivalently $\sigma_+^{-1}(i) = l$) means that rectangle i is the lth rectangle in σ_+. σ_- is similarly defined. In a feasible packing π, every pair i and j of rectangles satisfies at least one of the four conditions from (3) to (6). A sequence pair determines which of the four conditions is satisfied in the packing, as follows. Given a sequence pair $\sigma = (\sigma_+, \sigma_-)$, we define the partial orders $\preceq_\sigma^x$ and $\preceq_\sigma^y$ by

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

for any pair i and j of rectangles. Note that $i \preceq_\sigma^x i$ and $i \preceq_\sigma^y i$ hold for all i.

Property 1 [11]**:** Exactly one of the four relations $i \preceq_\sigma^x j$, $j \preceq_\sigma^x i$, $i \preceq_\sigma^y j$ and $j \preceq_\sigma^y i$ holds for any pair i and j of rectangles with $i \ne j$.

Then, given a sequence pair $\sigma = (\sigma_+, \sigma_-)$ and a vector of modes $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, we define $\Pi_{\sigma,\mu}$ as the set of packings π that satisfy the following three conditions for all i and $j \in I$:

$$\mu_i(\pi) = \mu_i,$$
$$i \preceq_\sigma^x j \text{ and } i \ne j \Longrightarrow x_i(\pi) + w_i^{(\mu_i)} \le x_j(\pi), \tag{7}$$
$$i \preceq_\sigma^y j \text{ and } i \ne j \Longrightarrow y_i(\pi) + h_i^{(\mu_i)} \le y_j(\pi). \tag{8}$$

This means that any packing $\pi \in \Pi_{\sigma,\mu}$ is feasible and satisfies the mode constraint. Conversely, it can be shown that, for any feasible packing π, there exists a pair of σ and μ that satisfies $\pi \in \Pi_{\sigma,\mu}$. It is shown in [11] that such a sequence pair $\sigma = (\sigma_+, \sigma_-)$ exists for any packing π, without considering the time complexity for obtaining it. The encoding algorithms in Appendix A (one of which runs in $O(n \log n)$ time) also give a proof of this fact.

3.2. Computing an optimal packing in $\Pi_{\sigma,\mu}$

In this section, we consider the following problem for a given (σ, μ):

RPGSC (σ, μ):	minimize	$g(p_{\max}(\pi), q_{\max}(\pi))$
	subject to	$\pi \in \Pi_{\sigma,\mu}$,

and propose a dynamic programming algorithm to compute an optimal packing π of RPGSC(σ, μ) in polynomial time. For simplicity, we omit the superscript representing the mode in Sections 3.2 to 3.4 (e.g., we use w_i instead of $w_i^{(\mu_i(\pi))}$), since the mode $\mu_i(\pi)$ of each rectangle i is fixed when we consider decoding algorithms. By Property 1, we can obtain a feasible packing even if we determine the horizontal and vertical coordinates separately. Moreover, since the objective function $g(p_{\max}(\pi), q_{\max}(\pi))$ is assumed to be nondecreasing in $p_{\max}(\pi)$ and $q_{\max}(\pi)$, respectively, it can be minimized by minimizing $p_{\max}(\pi)$ and $q_{\max}(\pi)$ independently. We will give below an algorithm to minimize $p_{\max}(\pi)$. An algorithm to minimize $q_{\max}(\pi)$ can be similarly defined.

Let us define J_i^{f} and $f_i(x)$ for each i as follows:

$J_i^{\mathrm{f}} = \{j \in I \mid j \preceq_\sigma^x i\}$,

$f_i(x)$: the minimum value of $\max_{j \in J_i^{\mathrm{f}}} p_j(x_j(\pi))$ subject to $x_j(\pi)+w_j \leq x_{j'}(\pi)$ for all $j, j' \in J_i^{\mathrm{f}}$ such that $j \neq j'$ and $j \preceq_\sigma^x j'$, and $x_i(\pi) + w_i \leq x$.

We call $f_i(x)$ the minimum penalty function. This function is nonincreasing in x by the definition, and the minimum penalty value $p_{\max}(\pi)$ of (1) can be obtained by

$$\max_{i \in I} \min_x f_i(x).$$

Then, by the idea of dynamic programming, $f_i(x)$ can be computed by

$$f_i(x) = \begin{cases} \min_{x_i \leq x - w_i} p_i(x_i), & \text{if } J_i^{\mathrm{f}} = \{i\}, \\ \min_{x_i \leq x - w_i} \max\{p_i(x_i), \max_{j \in J_i^{\mathrm{f}} \setminus \{i\}} f_j(x_i)\}, & \text{otherwise.} \end{cases} \tag{9}$$

The horizontal coordinates $x_i(\pi)$ of each rectangle i can be computed by

$$x_i(\pi) = \begin{cases} \max\{x_i \mid p_i(x_i) = \min_{x_i'}\{p_i(x_i') \mid f_i(x_i' + w_i) = \min_x f_i(x)\}\}, \\ \qquad \text{if } J_i^{\mathrm{b}} = \{i\}, \\ \max\{x_i \mid p_i(x_i) = \min_{x_i'}\{p_i(x_i') \mid f_i(x_i' + w_i) = \min_x \{f_i(x) \mid x \leq r_i\}\}\}, \\ \qquad \text{otherwise,} \end{cases} \tag{10}$$

where $J_i^{\mathrm{b}} = \{j \in I \mid i \preceq_\sigma^x j\}$ and $r_i = \min_{j \in J_i^{\mathrm{b}} \setminus \{i\}} x_j(\pi)$. We can minimize $p_{\max}(\pi)$ with these horizontal coordinates, and moreover, we can minimize $p_i(x_i(\pi))$ for all i locally.

In order to consider how to compute the above $f_i(x)$, let us define procedure Take-Min-Max (g_1, g_2) and Take-Max (g_1, g_2) for two functions g_1 and g_2 as follows.

Procedure Take-Min-Max (g_1, g_2)

Output function $g_{\mathrm{tmm}}(x) = \min_{t \leq x} \max\{g_1(t), g_2(t)\}$ and stop.

Procedure Take-Max (g_1, g_2)

Output function $g_{\mathrm{tm}}(x) = \max\{g_1(x), g_2(x)\}$ and stop.

These are basic operations to compute $f_i(x)$, and can be done in $O(\tau_1 + \tau_2)$ time in either case, where τ_1 and τ_2 are the space complexity (i.e., the number of linear pieces) of functions g_1 and g_2, respectively. In case g_1 and g_2 are nonincreasing, output functions by these two procedures become the same (i.e., $g_{\mathrm{tmm}}(x) = g_{\mathrm{tm}}(x)$), and this condition is always satisfied when we call procedure Take-Max in our decoding algorithms. Then, we use procedure Take-Min-Max instead of Take-Max.

If we compute $f_i(x)$ by using (9) naively, we call procedure Take-Min-Max $O(n^2)$ times since $\sum_i |J_i^{\mathrm{f}}| = O(n^2)$. However, we propose here a decoding algorithm in which we call procedure Take-Min-Max $O(n \log n)$ times in total.

In order to compute $f_i(x)$ for each $i = 1, 2, \ldots, n$, we should compute $\max_{j \in J_i^{\mathrm{f}} \setminus \{i\}} f_j(x_i)$ in (9). We introduce a complete binary tree of height $\lceil \log_2 n \rceil$ with n leaves. The leaves are labeled $1, 2, \ldots, n$ from left to right, where leaf $l \in \{1, 2, \ldots, n\}$ corresponds to rectangle $\sigma_-(l)$. We define a function $f_l^k(x)$ for each $l \in \{1, 2, \ldots, n\}$ and $k \in \{0, 1, \ldots, n\}$ as follows:

$$f_l^k(x) = \begin{cases} f_{\sigma_-(l)}(x), & \text{if } \sigma_+^{-1}(\sigma_-(l)) \leq k, \\ -\infty, & \text{if } \sigma_+^{-1}(\sigma_-(l)) \geq k+1. \end{cases} \tag{11}$$

Then, we can compute $f_i(x)$ of (9) from $p_i(x)$ and $f_l^k(x)$ since

$$\max_{j \in J_i^{\mathrm{f}} \setminus \{i\}} f_j(x) = \max\{f_l^k(x) \mid l < \sigma_-(i) \text{ and } k = \sigma_+(i) - 1\}. \tag{12}$$

To compute this efficiently, internal nodes of the binary tree are labeled by distinct numbers $l \geq n+1$ such as $n+2^d, n+2^d+1, \ldots, n+2^{d+1}-1$ from left to right for all nodes whose depths from the root node are d (i.e., the root node is labeled $n+1$ and the maximum number of node labels becomes $n+2^{(\lceil \log_2 n \rceil - 1)}$). Let T_l be the set of leaf labels in the subtree whose root node is l. We define a function $g_l^k(x)$ for each internal node l and $k \in \{0, 1, \ldots, n\}$ as follows:

$$g_l^k(x) = \max_{j \in T_l} f_j^k(x).$$

We compute $f_l^k(x)$ and $g_l^k(x)$ from $k = 1$ to n step by step. Initially, $f_l^0(x) = -\infty$ and $g_l^0(x) = -\infty$ hold for all leaves and internal nodes. Now consider a $k \in \{1, 2, \ldots, n\}$. First, we will compute $f_l^k(x)$ for all leaves l. In this step, $f_l^k(x) = f_l^{k-1}(x)$ holds for each l such that $\sigma_+^{-1}(\sigma_-(l)) \neq k$, and hence we should compute only $f_{l_k}^k(x)$ such that $\sigma_+^{-1}(\sigma_-(l_k)) = k$ (equivalently $l_k = \sigma_-^{-1}(\sigma_+(k))$). To compute this, we define $g^k(x)$. Let us initialize $g^k(x) := -\infty$ and then repeat the following step along the path in the tree from the root to leaf $\sigma_-^{-1}(\sigma_+(k))$ (we call this path RL-path, where RL stands for root to leaf): If the path goes from node l to its right child, then let $g^k(x) :=$ Take-Min-Max $(g_{l'}^{k-1}, g^k)$ (if l' is an internal node) or $g^k(x) :=$ Take-Min-Max $(f_{l'}^{k-1}, g^k)$ (if l' is a leaf) for the left child l' of l. Finally, $g^k(x)$ becomes $\max\{f_l^{k-1}(x) \mid l < \sigma_-^{-1}(\sigma_+(k))\}$ (see Fig. 2 as an example). This is equal to the right hand side of (12), and then we can compute $f_i(x)$ $(= f_{l_k}^k(x))$ by (9). Next, we will compute $g_l^k(x)$ for all internal nodes l. In this step, $g_l^k(x) = g_l^{k-1}(x)$ holds for each l such that $l_k \notin T_l$, and hence we should compute only $g_l^k(x)$ such that $l_k \in T_l$, and this condition means that node l is in RL-path. Then, we compute $g_l^k(x) :=$ Take-Min-Max $(g_l^{k-1}, f_{l_k}^k)$ for all internal nodes l in RL-path.

We call the above procedure to compute $f_i(x)$ for all i algorithm Compute-Minimum-Penalty-Function (CMPF), which is formally described as follows.

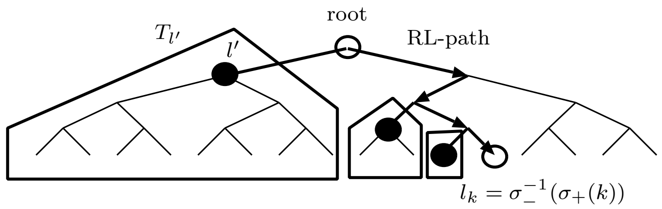


Fig. 2. An example to compute $g^k(x)$

Algorithm Compute-Minimum-Penalty-Function (CMPF)

Step 1 Make a complete binary tree of height $\lceil \log_2 n \rceil$ with n leaves. The leaves are labeled $1, 2, \ldots, n$ from left to right and internal nodes are labeled by distinct numbers more than n.
Step 2 Let $f_l^0(x) := -\infty$ for all leaves, $g_l^0(x) := -\infty$ for all internal nodes and $k := 1$.
Step 3 (Compute $g^k(x)$ along RL-path.)
  3.1 Let $g^k(x) := -\infty$ and l be the root node.
  3.2 If $l \in \{1, 2, \ldots, n\}$ (i.e., leaf), then go to Step 4;
  otherwise let l^* be the child node of l in RL-path.
  3.3 If l^* is the left child of l, then let $l := l^*$ and return to 3.2;
  otherwise let $g^k(x) :=$ Take-Min-Max $(g_{l'}^{k-1}, g^k)$ (l': internal node) or Take-Min-Max $(f_{l'}^{k-1}, g^k)$ (l': leaf) for the left child l' of l. Let $l := l^*$ and return to 3.2.
Step 4 Let $f_{\sigma_+(k)}(x)$ $(= f_{l_k}^k(x)) :=$ Take-Min-Max $(p_{\sigma_+(k)}, g^k)$.
Step 5 Let $g_l^k(x) :=$ Take-Min-Max $(g_l^{k-1}, f_{l_k}^k)$ for all internal nodes l in RL-path.
Step 6 If $k = n$, then output $f_i(x)$ for all i and stop; otherwise let $k := k + 1$ and return to Step 3.

Since procedure Take-Min-Max is called $O(\log n)$ times for each k (where $O(\log n)$ is the height of the tree), the total number of calls to Take-Min-Max is $O(n \log n)$.

Now, we evaluate the time complexity of algorithm CMPF. Let δ_i be the number of linear pieces of $p_i(x)$, and let $\delta = \sum_i \delta_i$. Since we are given the information of each linear piece of functions $p_i(x)$ explicitly (see the assumption in Section 2), δ becomes a lower bound of the input size. We define τ as an upper bound on the space complexity of functions $f_i(x)$, and τ also becomes an upper bound of $f_l^k(x)$ and $g_l^k(x)$ since $f_i(x)$ can be computed from $f_l^k(x)$ and $g_l^k(x)$. (More details of τ will be discussed in Section 3.3.) Thus the time complexity of algorithm CMPF is $O(\tau n \log n + \delta)$.

3.3. Time complexity of the decoding algorithm

In order to derive the time complexity of the decoding algorithm in the previous section, we first consider the space complexity of $f_i(x)$, $f_l^k(x)$ and $g_l^k(x)$. We consider the following three cases.

(1) Cost function $p_i(x)$ satisfies the following property: There exists a d_i that satisfies $p_i(x) = +\infty$ for $x < d_i$, and $p_i(x)$ is nondecreasing for $x \geq d_i$. Then τ becomes $O(1)$. In this case, we can find an optimal packing from a given sequence pair in $O(n \log n + \delta)$ time by algorithm CMPF. Many rectangle packing problems, such as those considered in [11,12,15,16], can be reduced to this case and furthermore satisfy $\delta = O(n)$, indicating that our algorithm CMPF runs in $O(n \log n)$ time. This time complexity is the same as those discussed in [15,16].

(2) Cost functions $p_i(x)$ are convex for all i. Then functions $f_l^k(x)$ and $g_l^k(x)$ are convex and nonincreasing, and have negative gradients that also appear in cost functions $p_i(x)$. Let ξ be the number of different values among the negative gradients in all cost functions $p_i(x)$. In this case, the maximum space complexity of $f_l^k(x)$ and $g_l^k(x)$ is $\xi + 1$, and hence the time complexity of algorithm CMPF becomes $O(\xi n \log n + \delta)$. In many applications, $\delta = O(n)$ and ξ can be regarded as a constant, and CMPF is quite efficient in such cases.

(3) Each $p_i(x)$ is general piecewise linear. That is, $p_i(x)$ can be non-convex and discontinuous. In this case, the space complexity of each $f_l^k(x)$ and $g_l^k(x)$ becomes $O(\delta\, \alpha(\delta, \delta))$, where $\alpha(m, n)$ is the inverse of Ackermann function. (It is known that $\alpha(m, n) \leq 4$ holds for most realistic values of m and n.) The reason is explained as follows. $f_l^k(x)$ and $g_l^k(x)$ can be represented as the maximum of some linear pieces of $p_i(x)$ and some pieces which are parallel to x-axis. The pieces of the latter type can be generated in the process of computing $f_l^k(x)$ and $g_l^k(x)$ from some pieces of $p_i(x)$ which have positive gradients or by some discontinuous points of $p_i(x)$ which have positive gaps. The total number of linear pieces of $p_i(x)$ is $O(\delta)$ and the number of pieces parallel to x-axis is also $O(\delta)$. Hence the space complexity of $f_l^k(x)$ and $g_l^k(x)$ is given by the space complexity of the upper envelope of $O(\delta)$ line segments, which is known to be $O(\delta\, \alpha(\delta, \delta))$ [1]. Therefore the time complexity of algorithm CMPF is $O(\delta \alpha(\delta, \delta) n \log n)$. In many realistic cases, $\delta_i = O(1)$ (i.e., $\delta = O(n)$) holds (and $\alpha(\delta, \delta)$ can be regarded as $O(1)$ as mentioned above), and hence the time complexity of CMPF becomes $O(n^2 \log n)$.

Remark. The time complexity of algorithm CMPF depends on δ for all three cases. Algorithm CMPF is a polynomial time algorithm under the assumption that the information of each linear piece of function $p_i^{(k)}(x)$ is given explicitly. However, in general, δ can be exponentially large if functions $p_i^{(k)}(x)$ are given implicitly, and in such cases, the algorithm proposed by Ahuja, Hochbaum and Orlin [2,3] is more efficient for case 2. Note that they consider a slightly different problem and a careful transformation is necessary.

3.4. An improved decoding algorithm

In this section, we describe an algorithm Compute-Better-Packing (CBP), which computes a packing π from a given (σ, μ). Although the packing π computed by CBP may not satisfy $\pi \in \Pi_{\sigma,\mu}$, it is not worse than any packing in $\Pi_{\sigma,\mu}$ in terms of the objective value. While algorithm CMPF in Section 3.2 computes the x and y coordinates of all rectangles independently, CBP computes the coordinates of one direction first, and then computes the coordinates of the other direction on the basis of the first coordinates. We assume here that CBP computes the y-coordinates first, since the other case is similar.

First we define a packing π^0. The y-coordinates of rectangles in π^0 are determined by applying algorithm CMPF, while the x-coordinates are given by

$$\begin{aligned}&x_{\sigma_-(1)}(\pi^0) = 0,\\ &x_{\sigma_-(l)}(\pi^0) = x_{\sigma_-(l-1)}(\pi^0) + w_{\sigma_-(l-1)},\ l = 2, 3, \ldots, n.\end{aligned}$$

The obtained packing π^0 is in $\Pi_{\sigma,\mu}$ and $q_{\max}(\pi^0) \leq q_{\max}(\pi)$ holds for all $\pi \in \Pi_{\sigma,\mu}$. Next, we define the set $I_i^{\mathrm{L}} \subseteq I$ (L stands for left) for each $i \in I$ as follows:

$$\begin{aligned} j \in I_i^{\mathrm{L}} \iff\ & x_j(\pi) + w_j \leq x_i(\pi),\ y_j(\pi) - h_i < y_i(\pi) < y_j(\pi) + h_j \text{ and the line}\\ & \text{segment } ((x_j(\pi) + w_j, y), (x_i(\pi), y)) \text{ does not cross any rectangle}\\ & \text{in } I, \text{ for some } y \text{ satisfying } \max\{y_i(\pi), y_j(\pi)\} < y < \min\{y_i(\pi)+h_i,\\ & y_j(\pi) + h_j\}.\end{aligned}$$

Note that we permit that the line segment $((x_j(\pi)+w_j, y), (x_i(\pi), y))$ has length 0. Fig. 3 illustrates set I_i^{L} and locations of rectangles. The set I_i^{L} can be computed

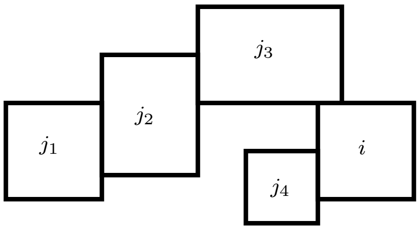


Fig. 3. An example of rectangles ($j_2, j_4 \in I_i^L$ and $j_1, j_3 \notin I_i^L$)

for all $i \in I$ in $O(n \log n)$ time by using the well-known plane sweep technique [4]. (We explain the algorithm to compute the set I_i^{L}s in Appendix A.) Then, we compute a packing π that minimizes $p_{\max}(\pi)$ among those satisfying

$$j \in I_i^{\mathrm{L}} \Longrightarrow x_j(\pi) + w_j \leq x_i(\pi) \text{ for all } i \in I, \tag{13}$$

where I_i^{L} is defined on π^0 and $y_i(\pi) = y_i(\pi^0)$ holds for all $i \in I$. Since $j \in I_i^{\mathrm{L}} \Longrightarrow j \preceq_\sigma^x i$ holds for all i and j $(i \neq j)$, constraints (13) are not stronger than (7). That is, the feasible region of the problem considered here contains that of RPGSC(σ,μ). Hence, the $p_{\max}(\pi)$ obtained by CBP is not worse than that obtained by CMPF. Let us define $\tilde{f}_i(x)$ as the minimum value of $\max_{j\in I_i^{\mathrm{L}}\cup\{i\}} p_j(x_j(\pi))$ under the constraints of $x_i(\pi)+w_i \le x$ and $x_j(\pi)+w_j \le x_i(\pi)$ for all $j \in I_i^{\mathrm{L}}$. Then, $\tilde{f}_i(x)$ can be computed by

$$\tilde{f}_i(x) = \begin{cases} \min_{x_i \le x-w_i} p_i(x_i), & \text{if } I_i^{\mathrm{L}} = \emptyset, \\ \min_{x_i \le x-w_i} \max\{p_i(x_i), \max_{j\in I_i^{\mathrm{L}}} \tilde{f}_j(x_i)\}, & \text{otherwise.} \end{cases} \tag{14}$$

The minimum penalty value $p_{\max}(\pi)$ and the horizontal coordinate x_i of each rectangle i can be computed by the similar computation as explained in Section 3.2. It is easy to show that $\sum_i |I_i^{\mathrm{L}}| = O(n)$ holds (see Appendix A for details), and hence procedure Take-Min-Max is called $O(n)$ times in the recursion of (14).

The packing π computed by algorithm CBP may not satisfy $\pi \in \Pi_{\sigma,\mu(\pi)}$, where (σ,μ) is the coded solution to which CBP is applied. In this case, we can find a sequence pair σ' which satisfies $\pi \in \Pi_{\sigma',\mu(\pi)}$ in $O(n\log n)$ time by applying the encoding algorithm called P2SP-2 proposed in Appendix A to π. In our computational experiments, we will apply algorithm P2SP-2 to a packing π, whenever it is computed by CBP and is better than the current packing, so that the local search can resume from a coded solution (σ',μ) satisfying $\pi \in \Pi_{\sigma',\mu(\pi)}$.

The time complexity of algorithm CBP (even if including encoding algorithm) is bounded by the time to call Take-Min-Max $O(n\log n)$ times to compute the y-coordinates first, which is the same as that of CMPF.

4. Local search of coded solutions (σ,μ)

In this section, we propose metaheuristic algorithms to find good coded solutions (σ,μ). As metaheuristic algorithms are based on local search, we explain the general framework of local search in Section 4.1. After giving a definition of critical paths in Section 4.2, we explain four types neighborhoods based on critical paths in Section 4.3. In Section 4.4, we explain frameworks of the proposed metaheuristic algorithms.

4.1. Local search

The local search (LS) starts from an initial solution (σ,μ) and repeats replacing (σ,μ) with a better solution in its *neighborhood* $N(\sigma,\mu)$ until no better solution is found in $N(\sigma,\mu)$, where $N(\sigma,\mu)$ is a set of solutions obtainable from (σ,μ) by slight perturbations (which will be defined later). A solution (σ,μ) is called *locally optimal*, if no better solution exists in $N(\sigma,\mu)$. The LS from an initial solution $(\sigma^{(0)},\mu^{(0)})$, in which neighborhood N is used and solutions are evaluated by a function *eval*, is described as follows.

Algorithm LS$(N,(\sigma^{(0)},\mu^{(0)}))$

Step 1 Let $\sigma := \sigma^{(0)}$ and $\mu := \mu^{(0)}$.

Step 2 If there is a feasible solution $(\sigma', \mu') \in N(\sigma, \mu)$ such that $eval(\sigma', \mu') < eval(\sigma, \mu)$, let $\sigma := \sigma'$, $\mu := \mu'$ and return to Step 2. Otherwise output (σ, μ) and stop.

The following ingredients must be specified in designing LS: Search space, neighborhood, move strategy, an initial solution and a function to evaluate solutions. In our algorithm, we use the search space which is the set of all coded solutions (σ, μ). We adopt first admissible move strategy (i.e., when we find a better solution in its neighborhood, we move to the solution immediately). A solution (σ, μ) is basically evaluated by the objective value of the packing π obtained by algorithm CMPF in Section 3.2 or algorithm CBP in Section 3.4. However, to break ties, we compute the following three values:

1. Objective value $g(p_{\max}(\pi), q_{\max}(\pi)) + c(\mu(\pi))$,
2. the number of rectangles i for which $p_i^{(\mu_i(\pi))}(x_i(\pi)) = p_{\max}(\pi)$ or $q_i^{(\mu_i(\pi))}(y_i(\pi)) = q_{\max}(\pi)$ holds,
3. $\sum_i (p_i^{(\mu_i(\pi))}(x_i(\pi)) + q_i^{(\mu_i(\pi))}(y_i(\pi)))$.

In each criterion, a packing which has smaller value is better. We define $eval(\sigma, \mu)$ as the vector of these three values in this order, and use the lexicographic order of $eval(\sigma, \mu)$ to compare two solutions (i.e., criterion 2 is used when solutions are equivalent in criterion 1, and criterion 3 is used when solutions are equivalent in criteria 1 and 2).

4.2. Critical paths

Critical paths are defined for both of the x and y directions. We explain the definition only for the x direction, as that for the y direction is similar.

Given a packing $\pi \in \Pi_{\sigma,\mu}$, define a directed graph $G = (V, E)$ and subsets $S, T, \tilde{S}, \tilde{T} \subseteq I$ as follows:

$$
\begin{aligned}
&V = I,\\
&(i, j) \in E \iff x_i(\pi) + w_i^{(\mu_i(\pi))} = x_j(\pi) \text{ and } i \preceq_\sigma^x j,\\
&S = \{i \in I \mid p_i(x_i(\pi) - \varepsilon) \geq p_{\max}(\pi) \text{ for an arbitrarily small } \varepsilon > 0\},\\
&\tilde{S} = \{i \in S \mid p_i(x_i(\pi)) = p_{\max}(\pi)\},\\
&T = \{i \in I \mid p_i(x_i(\pi) + \varepsilon) \geq p_{\max}(\pi) \text{ for an arbitrarily small } \varepsilon > 0\},\\
&\tilde{T} = \{i \in T \mid p_i(x_i(\pi)) = p_{\max}(\pi)\}.
\end{aligned}
$$

Note that $i \in S$ and $i \notin \tilde{S}$ may occur if spatial cost function $p_i(x)$ is discontinuous at $x_i(\pi)$. We then define a *critical path* as a directed path in G, whose initial vertex s is in S, final vertex t is in T and either $s \in \tilde{S}$ or $t \in \tilde{T}$ holds. For any packing π obtained by our decoding algorithms, both S and T are nonempty and there is at least one critical path for each direction. It is easy to find all pairs of rectangles which are adjacent in critical paths in $O(n^2)$ time. An algorithm to compute critical paths with the worst case time complexity $O(n^2)$

(but runs in $O(n)$ time in practice) was proposed in [7]. Critical paths have an important property: $p_{\max}(\pi)$ cannot be decreased without breaking all critical paths. Therefore we introduce neighborhoods, which are based on critical paths, in the next section.

4.3. Neighborhoods

The neighborhood is a very important factor that determines the effectiveness of local search. We use the following four types of neighborhoods, called swap, shift, swap* and change mode.

4.3.1. Swap neighborhood A swap is the operation of exchanging the positions of two rectangles i and j in σ_+ and/or σ_-. If a swap is applied to one (resp., both) of σ_+ and σ_-, then the operation is called a *single swap* (resp., *double swap*). The single swap (resp., double swap) neighborhood is defined to be the set of all solutions obtainable from the current solution by single swap (resp., double swap) operations. The swap neighborhood is the union of the single swap and double swap neighborhoods. The size of the single swap neighborhood is $n(n-1)$ (there are $n(n-1)/2$ pairs of rectangles and two permutations), while that of double swap neighborhood is $n(n-1)/2$. The double swap neighborhood has the following property: If two rectangles i and j are exchanged in both of σ_+ and σ_-, then the constraints related to the partial orders $\preceq_\sigma^x$ and $\preceq_\sigma^y$ of these two rectangles are entirely exchanged. We propose three methods to reduce the size of swap neighborhood, which will be computationally compared in Section 5.3. (We call the swap neighborhood without any reductions SwapAll.)

(1) We impose the condition that one of the two rectangles i in the swap operation satisfies $p_i^{(\mu_i(\pi))}(x_i(\pi)) = p_{\max}(\pi)$ or $q_i^{(\mu_i(\pi))}(y_i(\pi)) = q_{\max}(\pi)$. We call this neighborhood SwapMax. The size of SwapMax is extremely small, and the possibility of missing some improved solutions in the original swap neighborhood appears to be high.

(2) Critical paths are used to reduce the neighborhood size. In a swap of i and j, the critical paths containing neither i nor j will not be broken (and the objective value does not decrease). Therefore, we choose at least one rectangle in a swap from those rectangles in critical paths of either direction. Then, the size of the neighborhood is reduced from $O(n^2)$ to $O(cn)$, where c is the number of rectangles in critical paths.

(3) This method can only be applied to the double swap neighborhood. We restrict the pairs of rectangles i and j to those satisfying at least one of the following two conditions:

- i is in the critical path of x direction and j is not. $w_i^{(\mu_i(\pi))} > w_j^{(\mu_j(\pi))}$.
- i is in the critical path of y direction and j is not. $h_i^{(\mu_i(\pi))} > h_j^{(\mu_j(\pi))}$.

The second and third methods have the following property: The current solution is locally optimal (with respect to the objective value) in the original swap neighborhood if no improved solution is found in the reduced neighborhood. In this sense, we can reduce the size of swap neighborhood without sacrificing the solution quality. In our computational experiments, we use the union of two neighborhoods, the single swap neighborhood reduced by method (2) and the double swap neighborhood reduced by method (3), and we call this SwapCri.

4.3.2. Shift neighborhood A shift is the operation of shifting the position of one rectangle i into another position in σ_+ and/or σ_-. If the position is changed in one permutation (resp., both permutations), we call it a *single shift* (resp., *double shift*). The single shift (resp., double shift) neighborhood is defined to be the set of all solutions obtainable from the current solution by single shift (resp., double shift) operations. In the case of the double shift neighborhood, we limit the positions, where the shifted rectangle is inserted, to those determined by the following rule. Let i be the rectangle to be shifted. Then we choose one rectangle j $(\neq i)$ arbitrary and insert i before or after j in both σ_+ and σ_-. Intuitively, in the packing space, we move rectangle i to the position just to the left of, right of, above or below the chosen j by these restricted operations. The shift neighborhood is the union of these two neighborhoods, and the size of this neighborhood is $O(n^2)$. We call this neighborhood ShiftAll.

We restrict further the rectangles to be shifted by the following rules, which will be computationally compared in Section 5.3.

(1) We restrict the rectangles to those satisfying $p_i^{(\mu_i(\pi))}(x_i(\pi)) = p_{\max}(\pi)$ or $q_i^{(\mu_i(\pi))}(y_i(\pi)) = q_{\max}(\pi)$. We call this neighborhood ShiftMax.

(2) We shift only those rectangles located in critical paths. We call this neighborhood ShiftCri.

4.3.3. Swap neighborhood* A swap* operation breaks a critical path while preserving the relations $\preceq_\sigma^x$ and $\preceq_\sigma^y$ between other rectangles as much as possible. We explain a swap* operation only for the x direction. It removes two rectangles i and j, which are adjacent in the horizontal critical path, from σ_+ and/or σ_-, and inserts them into adjacent positions (but in the reverse order) of σ_+ and/or σ_- between the original positions of i and j (see Fig. 4 for illustration). This operation is formally defined as follows. Here, only the case of changing σ_+ to σ'_+ is explained. The operation on σ_- is similar. Let us assume that rectangles i and j are adjacent in a critical path, and $\sigma_+(\alpha) = i$ and $\sigma_+(\beta) = j$ hold in the current solution. Let γ be an integer that satisfies $\alpha \leq \gamma < \beta$. Then, the resulting permutation σ'_+ is given by (see Fig. 4) :

$$
\begin{aligned}
&\sigma'_+(l) = \sigma_+(l+1), l = \alpha, \ldots, \gamma - 1,\\
&\sigma'_+(\gamma) = j,\\
&\sigma'_+(\gamma+1) = i,\\
&\sigma'_+(l) = \sigma_+(l-1), l = \gamma+2, \ldots, \beta,\\
&\sigma'_+(l) = \sigma_+(l), l = 1, 2, \ldots, \alpha - 1, \beta + 1, \ldots, n.
\end{aligned}
$$

Then, we can change from a packing in Fig. 5 (a) to a packing in Fig. 5 (b)

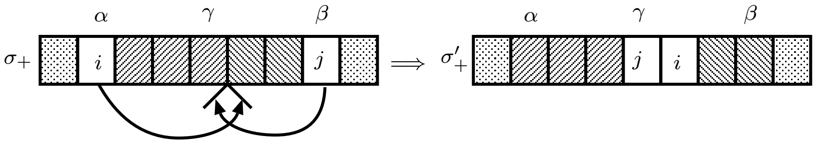


Fig. 4. An example of changing σ_+ to σ'_+

with a swap* operation on i and j. We consider all possible i and j, and all γ satisfying $\alpha \leq \gamma < \beta$; hence the neighborhood size is $O(n^3)$ in the worst case. In practice, however, the average size appears to be much smaller. We call this neighborhood Swap*.

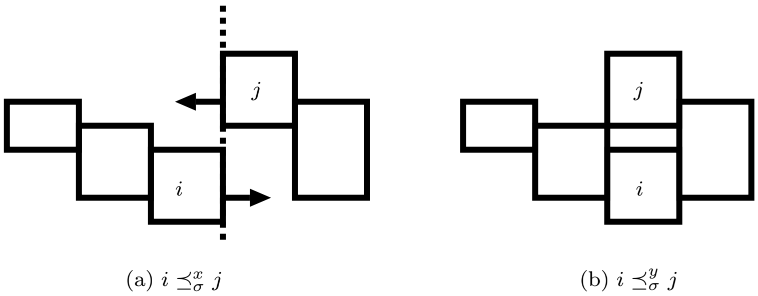


Fig. 5. An example showing the effect of a swap* operation

4.3.4. Change mode neighborhood The change mode neighborhood is the set of solutions obtainable from the current coded solution (σ, μ) by changing the mode $\mu_i(\pi)$ of one rectangle i. This is the only operation applied to the vector μ. The size of this neighborhood is $\sum_i (m_i - 1)$ where m_i is the number of modes of rectangle i.

4.3.5. Combination of some neighborhoods It is often effective to combine more than one neighborhood. In the computational experiment of Section 5, we use a combination of SwapCri, ShiftMax and Swap*, which is called Union. Moreover, we use the change mode neighborhood in combination with other neighborhoods whenever we treat a problem in which each rectangle has more than one mode, since this is the only operation applied to the mode vector μ. When we use more than one neighborhood, we use them in random order in our computational experiments.

4.4. Metaheuristics

We explain three metaheuristic algorithms, which are all based on local search. These will be used in our computational experiments in Section 5. Computation of these algorithms continues until the prespecified computational time is reached.

(1) The *random multi-start local search* (MLS). This is one of the simplest metaheuristic algorithms. In MLS, we randomly generate 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) [8]. ILS is a variant of MLS, in which initial solutions are generated by slightly perturbing a good solution $(\sigma_{\text{seed}}, \mu_{\text{seed}})$ found during the previous search. In our ILS, $(\sigma_{\text{seed}}, \mu_{\text{seed}})$ is defined to be the best solution obtained so far with respect to function *eval*. The next initial solution is then generated from $(\sigma_{\text{seed}}, \mu_{\text{seed}})$ by applying swap, shift or change mode operations a few times randomly.

(3) The algorithm WALK is similar to algorithm WALK-SAT [14] proposed for the satisfiability problem. We define neighborhood $N(\sigma, \mu, i)$ for a coded solution (σ, μ) and a rectangle i as follows: $N(\sigma, \mu, i)$ is the set of solutions obtainable from (σ, μ) by applying swap, shift, swap* or change mode operation to rectangle i. In WALK, we first choose a rectangle i randomly from those in the critical paths, and then choose the best solution (σ', μ') in $N(\sigma, \mu, i) \setminus \{(\sigma, \mu)\}$ and move to (σ', μ') (i.e., let $(\sigma, \mu) := (\sigma', \mu')$) even if the eval of (σ', μ') is worse than that of (σ, μ).

5. Computational experiment

In this section, our algorithms are evaluated on some instances of the rectangle packing and scheduling problems. The algorithms were coded in C language and run on a handmade PC (Intel Pentium III 1 GHz, 1 GB memory).

We describe test instances in Section 5.1. In Section 5.2, we examine the performance of our decoding algorithms proposed in Sections 3.2 and 3.4. We then report computational results of local search with various implementations using different neighborhoods in Section 5.3. Computational results of various metaheuristic algorithms are reported in Section 5.4. In Section 5.5, we compare our algorithms with other existing heuristic algorithms for both the rectangle packing problem and the scheduling problem.

5.1. Test problems and their instances

We explain two problems and their instances, which are used in our computational experiments. All instances can be obtained electronically from

`http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing/` .

5.1.1. The area minimization problem We are given a set of n rectangles $I = \{1, 2, \ldots, n\}$, where each rectangle $i \in I$ has a width w_i and a height h_i. The rotations of 90° are allowed and the objective is to minimize the area of the rectangle bin that contains all given rectangles. This problem was considered in many papers including [11,12,15,16]. In our formulation as RPGSC instances, each rectangle has two modes corresponding to its orientations: (1) the original orientation and (2) the orientation after 90° rotation. For each $i = 1, 2, \ldots, n$, we set

$$\begin{aligned} w_i^{(1)} &= w_i, & h_i^{(1)} &= h_i, \\ w_i^{(2)} &= h_i, & h_i^{(2)} &= w_i. \end{aligned}$$

For $i = 1, 2, \ldots, n$ and $k = 1, 2$, we use

$$p_i^{(k)}(x) = \begin{cases} +\infty & (x < 0) \\ x + w_i^{(k)} & (x \geq 0), \end{cases} \quad q_i^{(k)}(y) = \begin{cases} +\infty & (y < 0) \\ y + h_i^{(k)} & (y \geq 0). \end{cases}$$

The objective of the resulting RPGSC instance is to minimize $p_{\max}(\pi) \cdot q_{\max}(\pi)$ (i.e., the area of the rectangle that covers all given rectangles).

We use five instances of this problem: ami49, rp100, pcb146, rp200 and pcb500. The first instance ami49 has 49 rectangles, whose data are obtainable electronically from `http://www.cbl.ncsu.edu/CBL_Docs/lys90.html`. Instances rp100 and rp200 were randomly generated, and have 100 and 200 rectangles, respectively. The generation was done by randomly choosing integers from [1,100] for widths and heights of rectangles. Instances pcb146 and pcb500 were given by Kajitani [11,12]. These instances have 146 and 500 rectangles, respectively. Since the optimal solutions of these instances are unknown, we use the sum of the areas of n rectangles as a lower bound of the objective function.

5.1.2. The scheduling problem of large building blocks This is a problem encountered in a factory producing large building blocks. The blocks produced are very large, and each block stays in the same position until all the processes on it are finished. Each building block i has a length l_i, a processing time t_i, a ready time s_i and a due date d_i. As the shape of the work space is long and narrow, the building blocks can be regarded as one dimensional objects, which must be placed without overlap. A block i must arrive at the scheduled position after time s_i, and requires processing time t_i, before being removed prior to time d_i. A schedule is determined by the position, arrival time and removal time of each block. Let x_i be the start time of block i (hence its finish time is $x_i + t_i$). Then the objective is to minimize the $\max\{0, s_i - x_i, x_i + t_i - d_i\}$.

This problem can also be formulated as RPGSC, in which each rectangle has only one mode. For $i = 1, 2, \ldots, n$, let

$$\begin{aligned} & w_i^{(1)} = t_i, && h_i^{(1)} = l_i, \\ & p_i^{(1)}(x) = \begin{cases} -x + s_i & (x < s_i) \\ 0 & (s_i \leq x \leq d_i - t_i) \\ x + t_i - d_i & (x > d_i - t_i), \end{cases} && q_i^{(1)}(y) = \begin{cases} +\infty & (y < 0) \\ 0 & (0 \leq y \leq H - l_i) \\ +\infty & (y > H - l_i), \end{cases} \end{aligned}$$

Table 1. Computational time of the decoding algorithms in seconds

n	cost type	NAIVE	CMPF	CBP
49	special	5.04×10^{-5}	1.90×10^{-5}	6.41×10^{-5}
100	special	2.21×10^{-4}	4.56×10^{-5}	1.55×10^{-4}
146	special	4.56×10^{-4}	6.92×10^{-5}	2.34×10^{-4}
200	special	9.67×10^{-4}	9.61×10^{-5}	3.27×10^{-4}
500	special	5.63×10^{-3}	3.65×10^{-4}	1.21×10^{-3}
49	general	4.53×10^{-4}	1.85×10^{-4}	1.42×10^{-4}
100	general	2.11×10^{-3}	4.33×10^{-4}	3.46×10^{-4}
146	general	4.24×10^{-3}	9.51×10^{-4}	5.87×10^{-4}
200	general	1.03×10^{-2}	1.27×10^{-3}	8.85×10^{-4}
500	general	5.97×10^{-2}	6.13×10^{-3}	4.69×10^{-3}

where H is the length of the factory. The objective is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$. In this RPGSC formulation, the x-coordinate corresponds to time and the y-coordinate represents the positions of blocks. Given a packing π, $x_i(\pi)$ represents the start time and $y_i(\pi)$ represents the bottom edge of the position of block i.

We use five test instances sp78, sp50-a, sp50-b, sp100-a and sp100-b, where sp78 is a test instance from a real world application with 78 building blocks, and sp50-a, sp50-b, sp100-a, sp100-b are random instances that have 50 and 100 building blocks, respectively. It is known that there is a schedule with objective value 0 for every instance (i.e., no violation of constraints with respect to start time, due date and work space for all building blocks).

5.2. Decoding algorithms

In this section, we examine the performance of our decoding algorithms proposed in Sections 3.2 and 3.4. We compare three algorithms: (1) an $O(\tau n^2 + \delta)$ naive algorithm in Section 3.2 (denoted NAIVE), (2) algorithm CMPF in Section 3.2 whose time complexity is $O(\tau n \log n + \delta)$ and (3) algorithm CBP in Section 3.4 whose time complexity is the same as CMPF. Each algorithm is applied to the x-coordinate only. Note that we compute the y-coordinates of rectangles before applying CBP, since CBP can compute the x-coordinates in $O(n \log n + \tau n + \delta)$ time by making use of the y-coordinates of rectangles. We tested instances of several sizes and several cost functions. Results are shown in Table 1. The figures in the table show the computational time to obtain a packing from a given coded solution. Column "n" shows the size of instances and column "cost type" shows the type of cost functions. The type "special" means that all rectangles have cost functions described in Section 3.3-(1) (i.e., τ is $O(1)$), and "general" means that rectangles have general cost functions, each with a few segments.

From the table, we can observe that algorithms CMPF and CBP are more efficient than NAIVE. Moreover, CBP is faster than other algorithms for the instances with general cost functions, while it is about three times as slow as algorithm CMPF for the instances with special cost functions. This is because

Table 2. Quality of solutions of the decoding algorithms

	CMPF		CBP	
instance	value	time	value	time
ami49	93.37	1.44	93.30	2.20
rp100	94.06	15.47	94.85	27.04
pcb146	91.38	22.29	94.10	38.18
rp200	94.98	174.6	95.76	410.1
pcb500	93.49	3600.0	94.40	3600.0

CBP can have a larger coefficient on $n \log n$ term than CMPF, if τ is a small constant. This is because CBP uses the plane sweep algorithm, whose computational time is $O(n \log n)$, while CMPF does not. Here we emphasize that CBP is faster than CMPF even if the above general cost function on each rectangle has only a few segments. We also examined the performance of our decoding algorithms with respect to the quality of solutions. We use five test instances: ami49, rp100, pcb146, rp200 and pcb500. Algorithms CMPF and CBP are incorporated in the local search algorithms, and results are shown in Table 2. Column "value" shows the average of the following ratio, 100$\cdot$ (the objective value of the local optimum) / (the lower bound of the objective function), for ten trials (i.e., the larger the better). Column "time" shows the average computational time (in seconds) of local search algorithm LS. These notations are also used in Tables 3, 4 and 5. Note that we use neighborhood Union (defined in Section 4.3.5) in each LS. From the table, we can observe that CBP is superior to CMPF in quality for many instances. Based on these results, we will use CBP as our decoding algorithm in the experiments of Sections 5.3, 5.4 and 5.5.

5.3. Neighborhoods

The following eight types of neighborhoods discussed in Section 4.3 were computationally compared from the view point of the performance of the resulting local search algorithms LS, on test instances of the area minimizing problem in Section 5.1.1.

- SwapAll (the swap neighborhood in Section 4.3.1).
- SwapMax (the reduced swap neighborhood in Section 4.3.1).
- SwapCri (the reduced swap neighborhood in Section 4.3.1).
- ShiftAll (the shift neighborhood in Section 4.3.2).
- ShiftMax (the reduced shift neighborhood in Section 4.3.2).
- ShiftCri (the reduced shift neighborhood in Section 4.3.2).
- Swap* (the swap* neighborhood in Section 4.3.3).
- Union (the union of the SwapCri, ShiftMax and Swap* in Section 4.3.5).

Note that the change mode neighborhood is incorporated in all of the above eight cases.

Local search algorithms LS halt only when locally optimal solutions are obtained. To save computation time, however, we stop the search and output the

Table 3. Comparison of eight neighborhoods

	SwapAll		SwapMax		SwapCri	
instance	value	time	value	time	value	time
ami49	94.97	7.61	86.84	0.20	94.27	3.05
rp100	95.67	123.2	89.71	2.14	94.85	43.18
pcb146	95.57	457.3	87.67	8.32	93.64	39.39
rp200	95.73	1000.0†	90.87	19.51	95.57	808.2
pcb500	95.59	3600.0†	89.15	1190.0	95.55	3600.0†

	ShiftAll		ShiftMax		ShiftCri	
instance	value	time	value	time	value	time
ami49	93.25	11.03	84.89	0.22	91.81	1.65
rp100	93.65	178.5	86.60	2.17	92.57	25.65
pcb146	95.21	739.8	85.69	13.26	94.04	64.30
rp200	94.01	1000.0†	87.01	30.16	92.79	212.0
pcb500	93.70	3600.0†	86.92	1795.6	93.57	3600.0†

	Swap*		Union	
instance	value	time	value	time
ami49	83.43	0.12	93.30	2.20
rp100	86.56	1.29	94.85	27.04
pcb146	88.03	5.24	94.10	38.18
rp200	87.88	11.72	95.76	410.0
pcb500	89.96	625.8	94.40	3600.0†

current solution either when a locally optimal solution is obtained or when a prespecified computational time is reached. The time limit is 1000 seconds for instances with up to 200 rectangles, and is 3600 seconds for pcb500. Results are shown in Table 3 for five test instances. The mark "†" indicates that LS is forced to stop by the time limit. We can observe from Table 3 that SwapAll is the best neighborhood with respect to the quality of solutions, but takes much computational time. The solution quality of ShiftAll is also good, but is slightly worse than SwapAll, and its computational time is longer than SwapAll. Swap* and two restricted neighborhoods without using critical paths, SwapMax and ShiftMax, are very fast, but their solution quality is all poor. On the other hand, these using critical paths, SwapCri, ShiftCri and Union, show good performance with respect to both the quality of solutions and computational time, indicating that the use of critical paths is essential in reducing the neighborhood size effectively. Moreover, we can observe that Union is more effective than ShiftCri and SwapCri, and we will use Union in the experiments of Sections 5.4 and 5.5.

5.4. Metaheuristics

Next, we compared the three metaheuristic algorithms MLS, ILS and WALK described in Section 4.4, on five test instances, ami49, rp100, pcb146, rp200 and pcb500. We ran each algorithm until a prespecified computational time is reached. The time limit is 1000 seconds for instances with up to 200 rectangles,

Table 4. Comparison of three metaheuristic algorithms

	MLS		ILS		WALK	
instance	value	time	value	time	value	time
ami49	96.39	1000.0	96.96	1000.0	92.96	1000.0
rp100	95.95	1000.0	96.46	1000.0	94.13	1000.0
pcb146	95.47	1000.0	96.33	1000.0	91.82	1000.0
rp200	95.87	1000.0	96.10	1000.0	95.16	1000.0
pcb500	94.16	3600.0	94.16	3600.0	91.43	3600.0

Table 5. Comparison with other methods for the rectangle pacing problem

	SA-BSG		SA-SP		ILS	
instance	value	time	value	time	value	time
ami49	97.10	69.0	96.29	176.0	96.30	100.0
rp100	97.08	68.2	88.54	248.7	95.76	200.0
pcb146	94.87	100.2	94.42	678.7	95.63	300.0
rp200	N.A.	N.A.	N.A.	N.A.	95.67	400.0
pcb500	94.10	334.6	90.82	7802.9	92.27	1000.0

and is 3600 seconds for pcb500. Results are shown in Table 4. We can observe that ILS found better solutions than other two algorithms for many instances.

5.5. Comparison with other algorithms

Finally, we compared the performance of ILS with other existing heuristic algorithms, on various instances of the rectangle packing problem and the scheduling problem explained in Section 5.1. First, we compared our algorithm ILS with two heuristic algorithms for the area minimizing problem: (1) A simulated annealing algorithm with the BSG (bounded sliceline grid) coding scheme by Nakatake, Fujiyoshi, Murata and Kajitani (denoted SA-BSG) [12] and (2) a simulated annealing algorithm with the sequence pair coding scheme by Murata, Fujiyoshi, Nakatake and Kajitani (denoted SA-SP) [11]. We use five test instances of the area minimizing problem, ami49, rp100, pcb146, rp200 and pcb500. Note that algorithms (1) and (2) are specially tailored to the rectangle packing problem of minimizing the area, and the results of them for rp200 are not available (denoted N.A.). Results are shown in Table 5. From the table, we can observe that out algorithm is superior to SA-SP, but SA-BSG seems to be the best among three algorithms with respect to both the solution quality and the computational time. However, the difference in the quality is not large. It is emphasized that our algorithm is designed to solve more general problem, and can solve various types of packing and scheduling problems which can not be handled by SA-SP and SA-BSG. Taking this generality into consideration, the above results appear to be quite satisfactory.

Next, we compared the performance of our algorithm on instances of the scheduling problem of large building blocks explained in Section 5.1, with a

Table 6. Comparison with algorithm TS on the scheduling problem

	TS		ILS	
instance	ratio	time	ratio	time
sp50-a	10/10	394.8	10/10	44.98
sp50-b	1/10	2730.6	10/10	125.0
sp78	10/10	427.2	10/10	405.6
sp100-a	10/10	1851.9	10/10	362.6
sp100-b	9/10	1230.1	10/10	47.03

tabu search algorithm developed for the resource constrained project scheduling problem by Nonobe and Ibaraki (denoted TS) [13]. To solve the problem by TS, we reformulated the problem as a project scheduling problem in the way as described in [6]. Note that TS is not designed for solving the packing problem, but can handle more complicated scheduling problems with precedence constraints and other resources (e.g., manpower, machines and equipments) as well as work space. We stop the search either when an optimal solution (i.e., the objective value is 0) is obtained or when a prespecified computational time (we set the time limit to 3600 seconds) is reached. If a search stops after finding an optimum solution, it is called successful. We use the five instances described in Section 5.1, and results are shown in Table 6. Column "ratio" shows the (# of successful trials)/(# of trials). Column "time" shows the average computational time (in seconds) to find an optimal solution in successful trials. From the table, we can observe that ILS is superior to TS in both of the solution quality and the computational time for all instances. These results also exhibit good prospects of our algorithm.

6. Conclusion

In this paper, we introduced the rectangle packing problem with spatial costs, which is general in that it contains various types of packing problems and scheduling problems as special cases. We adopted the sequence pair as the coding scheme, which is a pair of permutations of the given n rectangles, and proposed decoding and encoding algorithms between coded solutions and packings. The decoding algorithm is based on dynamic programming and runs in $O(\tau n \log n + \delta)$ time, where τ and δ are the space complexity of the minimum penalty function and the spatial cost functions, respectively. This algorithm generalizes the results of [15, 16] in that it can deal with more general spatial costs, and runs in $O(n \log n)$ time, the same time complexity as those in [15, 16], if applied to the case of area minimization (considered in [11, 12, 15, 16]).

These algorithms were then incorporated in the local search and metaheuristic algorithms. We defined critical paths in a packing and proposed neighborhoods by making use of such critical paths. We conducted computational experiments and the results exhibited good prospects of the proposed algorithms.

Acknowledgment

The authors are grateful to the members of Kajitani's group, the University of Kitakyushu, for providing us the problem instances of the area minimization problem and their computational results, and Koji Nonobe, Kyoto University, for providing us his computational results. They are also grateful to Dorit S. Hochbaum, University of California, and Takeaki Uno, National Center of Science, for valuable comments. This research was partially supported by Scientific Grant-in-Aid, by the Ministry of Education, Culture, Sports, Science and Technology of Japan, and by the Telecommunications Advancement Foundation of Japan.

References

1. P.K. Agarwal, M. Sharir and P. Shor, Sharp upper and lower bounds on the length of general Davenport-Schinzel sequences, J. Comb. Theory, Ser. A, **52,** (1989) 228–274.
2. R.K. Ahuja, D.S. Hochbaum and J.B. Orlin, A cut-based algorithm for the non-linear dual of the minimum cost network flow problem, manuscript available at `http://web.mit.edu/jorlin/www/working_papers.html`.
3. R.K. Ahuja, D.S. Hochbaum and J.B. Orlin, Solving the convex cost integer dual network flow problem, G. Cornuejols, R.E. Burkard and G.J. Woeginger, eds., *Proc. IPCO'99, Lect. Notes Comput. Sci.* **1610** (Springer, 1999) 31–44.
4. M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf *Computational Geometry* (Springer, 1997).
5. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, 1979).
6. S. Hartmann, Packing problem and project scheduling models: an integrating perspective, J. Oper. Res. Soc. **51,** (2000) 1083–1092.
7. S. Imahori, Local search heuristics for the rectangle packing problem with general spatial costs, Master thesis, Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, 2001.
8. D.S. Johnson, Local optimization and the traveling salesman problem, M.S. Peterson, ed., *Automata, Languages and Programming, Lect. Notes Comput. Sci.* **443** (Springer, 1990) 446–461.
9. D. Liu and H. Teng, An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles, Eur. J. Oper. Res. **112,** (1999) 413–420.
10. A. Lodi, S. Martello, D. Vigo, Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems, Informs J. Comput. **11,** (1999) 345–357.
11. H. Murata, K. Fujiyoshi, S. Nakatake and Y. Kajitani, VLSI module placement based on rectangle-packing by the sequence-pair, IEEE Trans. Comput. Aided Des. **15-12,** (1996) 1518–1524.
12. S. Nakatake, K. Fujiyoshi, H. Murata and Y. Kajitani, Module placement on BSG-structure and IC layout applications, Proc. Intl. Conf. Comput. Aided Des. 1996, 484–491.
13. K. Nonobe and T. Ibaraki, Formulation and tabu search algorithm for the resource constrained project scheduling problem, *Essays and Surveys in Metaheuristics* (Kluwer Academic Publishers, 2001) 557–588.
14. B. Selman, H.A. Kautz and B. Cohen, Noise strategies for improving local search, Proc. 12th National Conf. Artif. Intell. 1994, 337–343.
15. T. Takahashi, 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,** (1996) 31–35.
16. X. Tang, R. Tian and D.F. Wong, Fast evaluation of sequence pair in block placement by longest common subsequence computation, Proc. Des. Autom. Test Eur. Conf. 2000, 106–111.
17. X. Tang and D.F. Wong, Fast-SP: a fast algorithm for block placement based on sequence pair, Proc. Asia S. Pac. Des. Autom. Conf. 2001, 521–526.

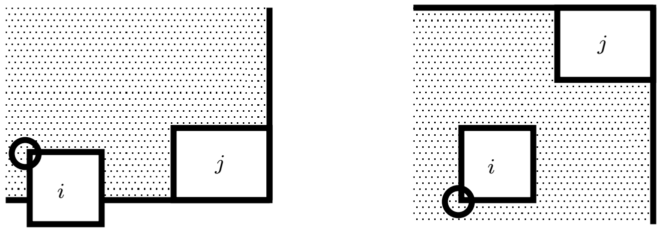
A. Transformation from packing to sequence pair

We propose an $O(n \log n)$ time algorithm to find a sequence pair $\sigma = (\sigma_+, \sigma_-)$ that satisfies $\pi \in \Pi_{\sigma,\mu(\pi)}$ for a given packing π and a mode vector $\mu(\pi)$. This problem is independent of the spatial cost functions. For simplicity, we omit the superscript representing the mode, since the mode $\mu_i(\pi)$ of each rectangle i is fixed when we consider this problem.

Based on a packing π, we define binary relations $\prec_+$ and $\prec_-$ on I as follows:

$$(x_i(\pi) < x_j(\pi) + w_j \text{ and } y_i(\pi) + h_i > y_j(\pi)) \Longleftrightarrow i \prec_+ j,$$
$$(x_i(\pi) < x_j(\pi) + w_j \text{ and } y_i(\pi) < y_j(\pi) + h_j) \Longleftrightarrow i \prec_- j.$$

Fig. 6 illustrates relationships $\prec_+$ and $\prec_-$ between nonoverlapping rectangles i and j. Relation $i \prec_+ j$ (resp., $i \prec_- j$) holds if and only if the upper left (resp., lower left) corner of rectangle i lies in the shaded area. Then let $\sigma = (\sigma_+, \sigma_-)$



(a) $i \prec_+ j$ (b) $i \prec_- j$

Fig. 6. Relationships between $\prec_+, \prec_-$ and coordinates of rectangles

be a sequence pair that satisfies the following two conditions:

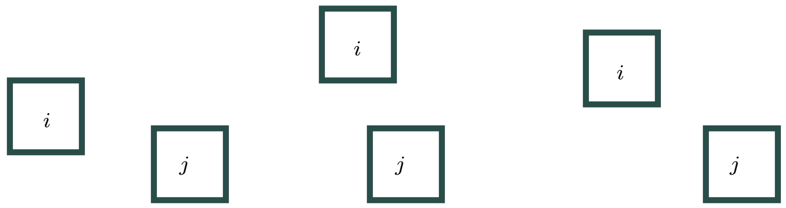
$$i \prec_+ j \Longrightarrow \sigma_+^{-1}(i) < \sigma_+^{-1}(j), \tag{15}$$
$$i \prec_- j \Longrightarrow \sigma_-^{-1}(i) < \sigma_-^{-1}(j). \tag{16}$$

Then we have the following lemma.

Lemma 1 A packing π satisfies $\pi \in \Pi_{\sigma,\mu(\pi)}$ for any sequence pair $\sigma = (\sigma_+, \sigma_-)$ satisfying conditions (15) and (16).

Proof: If nonoverlapping rectangles i and j in π are comparable in both relations $\prec_+$ and $\prec_-$, we note that exactly one of the four conditions (3)–(6) holds for i and j. For example, if $i \prec_+ j$ and $i \prec_- j$ hold, then the locations of i and j satisfy $x_i(\pi) + w_i \le x_j(\pi)$ (see Fig. 7 (a)). In this case, $\sigma = (\sigma_+, \sigma_-)$ satisfies $\sigma_+^{-1}(i) < \sigma_+^{-1}(j)$ and $\sigma_-^{-1}(i) < \sigma_-^{-1}(j)$ by (15) and (16), and π satisfies the implied condition (7) (i.e., $x_i(\pi) + w_i \le x_j(\pi)$). The case with $j \prec_+ i$ and $i \prec_- j$ is similar (see Fig. 7 (b)), and π satisfies the implied condition (8) (i.e., $y_i(\pi) + h_i \le y_j(\pi)$). If rectangles i and j are comparable in exactly one of relations $\prec_+$ and $\prec_-$, two of the four conditions (3)–(6) hold for i and j. For example, if i and j satisfies $i \prec_+ j$ but are not comparable in $\prec_-$ (i.e., $i \not\prec_- j$ and $j \not\prec_- j$), then both $x_i(\pi) + w_i \leq x_j(\pi)$ and $y_j(\pi) + h_j \leq y_i(\pi)$ hold (see Fig. 7 (c)). In this case, $\sigma = (\sigma_+, \sigma_-)$ satisfies $\sigma_+^{-1}(i) < \sigma_+^{-1}(j)$ by (15) but no restriction is imposed between $\sigma_-^{-1}(i)$ and $\sigma_-^{-1}(j)$. However, since π satisfies both the implied conditions (7) and (8) (corresponding to $\sigma_-^{-1}(i) < \sigma_-^{-1}(j)$ and $\sigma_-^{-1}(i) > \sigma_-^{-1}(j)$, respectively), π does not contradict with conditions (7) and (8). These arguments prove that of $\pi \in \Pi_{\sigma,\mu(\pi)}$ holds. ■

We first explain a simple $O(n^2)$ time algorithm to compute a sequence pair



(a) $i \prec_+ j$ and $i \prec_- j$ (b) $i \prec_+ j$ and $j \prec_- i$ (c) $i \prec_+ j$

Fig. 7. Relationships between packing π and relations $\prec_+$, $\prec_-$

$\sigma = (\sigma_+, \sigma_-)$ satisfying (15) and (16) for a given packing π, since the $O(n \log n)$ time algorithm is based on it. We describe only the case of σ_+ as the algorithm for σ_- is similar. We define sets $I_i^{\mathrm{LA}} = \{j \mid j \prec_+ i\}$ and $I_i^{\mathrm{RB}} = \{j \mid i \prec_+ j\}$ (LA (resp., RB) stands for left above (resp., right bottom)). Then, it is obvious that $j \in I_i^{\mathrm{LA}} \iff i \in I_j^{\mathrm{RB}}$ and $I_i^{\mathrm{LA}} \cap I_i^{\mathrm{RB}} = \emptyset$. For any subset $S \subseteq I$, at least one rectangle i satisfies $I_i^{\mathrm{LA}} \cap S = \emptyset$ (implying that no $j \in S$ satisfies $j \prec_+ i$). The $O(n^2)$ time algorithm to compute a sequence σ_+ is formally described as follows.

Algorithm Packing-to-Sequence-Pair-1 (P2SP-1)

Step 1 Compute I_i^{LA} for all rectangles $i \in I$. Let $S := I$ and $l := 1$.
Step 2 Choose a rectangle i such that $S \cap I_i^{\mathrm{LA}} = \emptyset$. Let $\sigma_+(l) := i$ and $S := S \setminus \{i\}$.
Step 3 If $S = \emptyset$, then output σ_+ and stop; otherwise let $l := l + 1$ and return to Step 2.

The time required in Step 1 is $O(n^2)$. To find an i satisfying $S \cap I_i^{\mathrm{LA}} = \emptyset$ in $O(n)$ time in Step 2, we keep the values of $|S \cap I_j^{\mathrm{LA}}|$ in memory for all $j \in S$. Such data can be maintained if we decrease $|S \cap I_j^{\mathrm{LA}}|$ by one for each $j \in S \cap I_i^{\mathrm{RB}}$ after i is removed from S. The loop of Steps 2 and 3 is repeated n times; hence the total time complexity is $O(n^2)$.

Now, we improve the above algorithm into an $O(n \log n)$ time algorithm. We use the set $I_i^{\mathrm{L}} \subseteq I$ for each $i \in I$ defined in Section 3.4. We also define

I_i^{R}, I_i^{B} and I_i^{A} similarly, where labels R, B and A stand for right, below and above, respectively. The sets I_i^{L}, I_i^{R}, I_i^{B} and I_i^{A} can be computed for all $i \in I$ in $O(n \log n)$ time by using the well-known plane sweep technique [4].

Though plane sweep is a standard technique, we explain its outline to show $\sum_{i \in I}(|I_i^{\mathrm{L}}| + |I_i^{\mathrm{R}}| + |I_i^{\mathrm{B}}| + |I_i^{\mathrm{A}}|) = O(n)$. To compute I_i^{L} for all i, we consider a sweep line parallel to the x-axis and move it from bottom to top. Let Q maintain the set of rectangles located on the sweep line during the sweep process. A rectangle i is inserted into (resp., deleted from) Q when the y-coordinate of the sweep line becomes $y_i(\pi)$ (resp., $y_i(\pi) + h_i$), hence set Q changes $2n$ times. Note that, if $y_i(\pi) = y_j(\pi) + h_j$ holds, we understand that j is deleted from Q before i is inserted into Q. I_i^{L} is initially set empty for all $i \in I$. When i is inserted into Q, we find the rectangle i^{L} (resp., i^{R}) in Q immediately to the left (resp., right) of i on the sweep line, and let $I_i^{\mathrm{L}} := I_i^{\mathrm{L}} \cup \{i^{\mathrm{L}}\}$ and $I_{i^{\mathrm{R}}}^{\mathrm{L}} := I_{i^{\mathrm{R}}}^{\mathrm{L}} \cup \{i\}$. When i is deleted from Q, we find the rectangle $i^{\mathrm{L}'}$ (resp., $i^{\mathrm{R}'}$) in Q immediately to the left (resp., right) of i just before i is deleted, and let $I_{i^{\mathrm{R}'}}^{\mathrm{L}} := I_{i^{\mathrm{R}'}}^{\mathrm{L}} \cup \{i^{\mathrm{L}'}\}$. (Some of i^{L}, i^{R}, $i^{\mathrm{L}'}$ and $i^{\mathrm{R}'}$ may not exist. In such cases, the corresponding operations are omitted.) For each i, j such that $j \in I_i^{\mathrm{L}}$, rectangle j becomes immediately to the left of rectangle i on the sweep line at least once. We set $I_i^{\mathrm{L}} := I_i^{\mathrm{L}} \cup j$ when j becomes immediately to the left of i for the first time; hence we can find all rectangles $j \in I_i^{\mathrm{L}}$ for each i with this operation.

In the above process, note that $\sum_i |I_i^{\mathrm{L}}|$ increases at most 2 whenever Q is changed, and hence $\sum_i |I_i^{\mathrm{L}}| = O(n)$. Thus the complexity to compute I_i^{L} for all i is $O(n \log n)$, since Q can be updated in $O(\log n)$ time when i is inserted and deleted, respectively, if an appropriate data structure such as the balanced search tree is used to keep Q [4]. Similarly, each of $\sum_i |I_i^{\mathrm{R}}|$, $\sum_i |I_i^{\mathrm{B}}|$ and $\sum_i |I_i^{\mathrm{A}}|$ is $O(n)$, and the time complexity of computing I_i^{R}, I_i^{B} and I_i^{A} for all i is $O(n \log n)$.

Let us consider relationship between I_i^{LA} and $I_i^{\mathrm{L}}, I_i^{\mathrm{A}}$. Given a set $S \subseteq I$, our objective is to find a rectangle i satisfying $I_i^{\mathrm{LA}} \cap S = \emptyset$. Let $S_{\mathrm{L,A}}^0$ be the set of rectangles $i \in S$ such that $(I_i^{\mathrm{L}} \cup I_i^{\mathrm{A}}) \cap S = \emptyset$ (for any subset $S \subseteq I$, at least one rectangle i satisfies $(I_i^{\mathrm{L}} \cup I_i^{\mathrm{A}}) \cap S = \emptyset$), and let $z(i) = \alpha \cdot y_i(\pi) - \beta \cdot x_i(\pi)$ for all $i \in I$ (α and β are nonnegative constants such that at least one of them is positive). Then, $I_i^{\mathrm{LA}} \cap S = \emptyset$ holds if $i \in S_{\mathrm{L,A}}^0$ and $z(i) \geq z(j)$ holds for all $j \in S_{\mathrm{L,A}}^0$. The algorithm to compute a sequence σ_+ in $O(n \log n)$ time is now described as follows.

Algorithm Packing-to-Sequence-Pair-2 (P2SP-2)

Step 1 Compute I_i^{L}, I_i^{R}, I_i^{B}, I_i^{A} and $z(i)$ for all rectangles $i \in I$.
Step 2 Let $S := I$ and $l := 1$. Compute $S_{\mathrm{L,A}}^0$.
Step 3 Choose a rectangle $i \in S_{\mathrm{L,A}}^0$ with the largest $z(i)$. Let $\sigma_+(l) := i$ and $S := S \setminus \{i\}$. Update $S_{\mathrm{L,A}}^0$.
Step 4 If $S = \emptyset$ holds, then output σ_+ and stop; otherwise let $l := l + 1$ and return to Step 3.

As mentioned above, Step 1 is possible in $O(n \log n)$ time by using the plane sweep technique. In Step 3, we choose a rectangle i with the maximum $z(i)$ among

$S^0_{\mathrm{L,A}}$ and delete it from S and $S^0_{\mathrm{L,A}}$. This is possible in $O(\log n)$ time if we use the data structure of heap to keep $S^0_{\mathrm{L,A}}$. We keep the values of $|(I^{\mathrm{L}}_j \cup I^{\mathrm{A}}_j) \cap S|$ in memory for all rectangles j and decrease $|(I^{\mathrm{L}}_j \cup I^{\mathrm{A}}_j) \cap S|$ by one for each $j \in (I^{\mathrm{R}}_i \cup I^{\mathrm{B}}_i) \cap S$ after i is removed from S in Step 3. Since $\sum_{i \in I}(|I^{\mathrm{R}}_i| + |I^{\mathrm{B}}_i|)$ is $O(n)$, this task of updating $|(I^{\mathrm{L}}_j \cup I^{\mathrm{A}}_j) \cap S|$ can be done in $O(n)$ time in total. Each rectangle i is inserted into $S^0_{\mathrm{L,A}}$ only once when $|(I^{\mathrm{L}}_i \cup I^{\mathrm{A}}_i) \cap S|$ becomes 0; hence the number of insertions (and deletions) is $O(n)$ and an insertion to $S^0_{\mathrm{L,A}}$ is also possible in $O(\log n)$ time. In summary, the total computational time of this algorithm is $O(n \log n)$.

Finally, we compared two algorithms (1) P2SP-1 (an $O(n^2)$ time encoding algorithm) and (2) P2SP-2 (an $O(n \log n)$ time encoding algorithm) by applying them to various instances in Section 5.1. The detailed results are omitted, but we could observe a significant speed up of P2SP-2 even for small instances such as ami49. Therefore we exclusively used P2SP-2 in the experiments in Sections 5.3, 5.4 and 5.5.