Studies on
Local Search Algorithms
for Cutting and Packing Problems

Shinji Imahori

Department of Applied Mathematics and Physics
Graduate School of Informatics
Kyoto University
Kyoto 606-8501, Japan

March, 2004

Doctoral Dissertation

submitted to Graduate School of Informatics, Kyoto University

in partial fulfillment of the requirement for the degree of

DOCTOR OF INFORMATICS

(Applied Mathematics and Physics)

Preface

A variety of combinatorial optimization problems appear in many application fields, and various algorithms have been proposed so far. However, there are numerous combinatorial optimization problems for which no polynomial time algorithm to find an optimal solution is known, e.g., those problems known as *NP-hard*. Under the widely believed conjecture P $\neq$ NP, exact algorithms for those problems must be exhaustively time consuming. To overcome this difficulty, we may make use of a fact that, in most practical applications, we do not need exact optimal solutions and are satisfied with sufficiently good solutions. In this sense, *approximate* or *heuristic algorithms*, which provide reasonably good solutions in practically meaningful time, are very important and have been intensively studied recently.

There are several useful tools to design approximate algorithms, such as *greedy method* and *local search*. These methods have been applied to many intractable problems for their simplicity and flexibility, and succeeded in obtaining good solutions to some extent. In recent years, more sophisticated algorithms that utilize these tools and some other techniques in more flexible frameworks such as *multi-start local search, genetic algorithm, simulated annealing, tabu search* and their variants have been studied, and applied to many NP-hard problems. Such algorithms are generically called *metaheuristics*.

In this thesis, we consider *cutting and packing problems* which are among such intractable problems. Cutting and Packing problems are important in numerous real-world applications, and hence many practical approximate algorithms have been proposed. However, there are numerous variants of cutting and packing problems, and many people, in particular those in application fields, cannot solve their variant with existing algorithms, nor afford to invest sufficient manpower and time to develop individual algorithms. The aim of the thesis is to propose general models that can include various types of specific variants, and to develop practical algorithms for them.

We first propose a new problem called the rectangle packing problem with general spatial costs (RPGSC). Introducing various cost functions and defining them appropriately, many types of cutting and packing problems and scheduling problems can be formulated in this form. For this problem, we develop practical approximate algorithms based on local

search and conduct thorough computational experiments with various type problems. We then propose another problem called the two-dimensional cutting stock problem with a constraint on the number of cutting patterns, and design local search algorithms based on various heuristics and mathematical programming techniques.

Cutting and packing problems are closely related to real-world applications in such fields as manufacturing industry, and new variants of the problems are continuously arising. It would be impossible to develop individual algorithms for all of such problems, and we believe it is very important to develop practical algorithms which can handle a wide range of specific problems. The author hopes that the work contained in this thesis will be helpful to advance the study in this intractable, important and interesting field.

March, 2004
Shinji Imahori

Acknowledgment

This thesis would not have been possible without the help of many people, whom I would like to acknowledge here.

First of all, I am heartily grateful to Professor Toshihide Ibaraki of Kyoto University for his enthusiastic guidance, discussion and persistent encouragement. He commented in detail on the whole work in the manuscript, which significantly improved the accuracy of the arguments and the quality of the expression. Without his considerable help, none of this work could have been completed.

I am deeply grateful to Professor Mutsunori Yagiura of Kyoto University. His heartful and earnest guidance has encouraged me all the time. Several enthusiastic discussions with him were quite exciting and invaluable experience for me.

I would like to express my sincere appreciation to Professor Xiaotie Deng of City University of Hong Kong, who guided me in research and gave me many valuable opportunities during my stay at his laboratory. I am indebted to Professor Koji Nonobe of Kyoto University and Professor Shunji Umetani of Toyota Technological Institute for numbers of helpful comments and suggestions.

I wish to express my gratitude to Professor Hiroshi Nagamochi of Toyohashi University of Technology, Professor Takeaki Uno of National Institute of Informatics, Professor Toshimasa Ishii of Toyohashi University of Technology, Professor Hirotaka Ono of Kyushu University, Professor Liang Zhao of Utsunomiya University, and all members in Professor Ibaraki's laboratory for many enlightening discussions on the area of this work and their warm friendship in the period I studied at Kyoto University.

I am also thankful to Professor Masao Fukushima of Kyoto University and Professor Yutaka Takahashi of Kyoto University for serving on my dissertation committee.

Finally, but not least, I would like to express my heartiest gratitude to my family for their heartfelt cooperation and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Combinatorial optimization problem

The optimization problem we consider in this thesis is generally defined as follows:

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && x \in F, \end{aligned} \tag{1.1.1}$$

where F denotes a set of solutions x that satisfy all the given constraints. F is called the *feasible region* and each $x \in F$ is called a *feasible solution*. The function f is called the *objective function*, and a feasible solution $x^* \in F$ is optimal if $f(x^*) \leq f(x)$ holds for all feasible solutions $x \in F$, and $f(x^*)$ is called the *optimal value*. We call an optimization problem (1.1.1) as a *combinatorial optimization problem* if F is combinatorial in some sense.

A variety of combinatorial optimization problems appear in many application fields; e.g., vehicle routing [2, 61], machine scheduling [14, 45], channel assignment [41, 67], timetabling [15, 74], plant location [26, 113], VLSI design [35, 89], crew scheduling [17, 72], production planning [86, 99] and so on, and numerous algorithms have been proposed so far. In the theory of computational complexity, an algorithm is generally regarded as efficient if its time complexity function is bounded above by a polynomial of the input length. For some combinatorial optimization problems, efficient algorithms which run in polynomial time have been proposed. For example, Kruskal [75] and Prim [98] proposed algorithms for the minimum spanning tree problem which run in $O(|E| \log |V|)$ time, and Dijkstra [27] proposed an algorithm for the shortest path problem which also runs in $O(|E| \log |V|)$ time, where V is a set of vertices and E is a set of edges.

On the other hand, there are many combinatorial optimization problems for which no polynomial time algorithm to find an optimal solution is known yet. Some examples of such problems are given in the following.

Traveling salesman problem

input: A complete directed graph $G = (V, E)$ and a cost function $c : E \to R_+$ (the set of nonnegative real number).

output: A minimum cost tour of G, i.e., a directed simple cycle of $|V|$ vertices with minimum total cost.

Vertex cover problem

input: An undirected graph $G = (V, E)$.

output: A minimum vertex cover of G, i.e., a subset $V' \subseteq V$ with minimum cardinality $|V'|$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V'.

Set covering problem

input: A finite set S and a collection $C = \{S_1, S_2, \ldots, S_m\}$ of subsets $S_i \subseteq S$.

output: A minimum set cover for S, i.e., a collection $C' \subseteq C$ of the minimum cardinality $|C'|$ such that every element in S belongs to at least one of $S_i \in C'$.

Graph coloring problem

input: An undirected graph $G = (V, E)$.

output: A coloring $\pi : V \to \{1, 2, \ldots, \chi\}$ of all vertices $v_i \in V$ with the minimum number of colors $|\chi|$, such that for each edge $\{u, v\} \in E$ vertices u and v have different colors.

Most of those problems are known to be *NP-hard* (actually, all of the above problems are NP-hard), and it is strongly believed that NP-hard problems have no polynomial time algorithm [39]. In other words, solving these problems exactly may necessitate enumerating an essential portion of the set of all solutions, whose number increases exponentially as problem size grows.

Since the NP-hardness is based on the worst case complexity, it may be possible to solve NP-hard problems efficiently in the practical sense. Representative methods frequently applied to this end are *branch-and-bound* and *dynamic programming*, which enumerate only promising solutions efficiently [60]. With intensive studies on these exact algorithms and as a result of the rapid progress of computer technology, the problem size that can be exactly solved has been increasing. However, it is not still large enough to accommodate all the problems arising in real applications.

Fortunately, in most applications, we are satisfied with good solutions obtained in reasonable computational time even if we are not able to obtain an exact optimal solution. In this sense, to deal with large instances of such intractable combinatorial optimization problems, *approximate* or *heuristic* algorithms are important. Actually, such approaches have been intensively studied in recent years.

In this thesis, we consider a certain type of intractable combinatorial optimization problems, called *cutting and packing problems*. These problems are not only being representative combinatorial optimization problems, but also important for industrial applications; e.g., wood, glass, steel and cloth industries, VLSI designing, newspaper paging, container loading and so on. In Section 1.2, we explain various types of cutting and packing problems, and show several algorithms for those problems. It is very hard to find an optimal solution for these problems, and hence we solve them heuristically. In Section 1.3, we show several useful tools to design approximate and heuristic algorithms.

1.2 Cutting and packing problems

In this section, we first explain cutting and packing problems in general. Given many small items, we want to place them into large objects without mutual overlap to minimize/maximize a given objective function. There are numerous problems which belong to this category, "cutting and packing problems," and there have been many studies since ancient days. In early stages, some problems of this category were solved by geometricians and puzzlers. They solved various problems by elementary geometry, experience, hunch, trial and error. Here we explain one of the problems that was considered from early days.

Packing of congruent circles in square

input: k circles of radius one.

output: A placement of circles without mutual overlap so that the area of the square that covers all the circles is minimized.

In many real-world applications, we encounter cutting and packing problems and they have been studied in numerous research disciplines such as computer science, operational research, engineering, manufacturing and others. As a result, many variants of cutting and packing problems are referred to in these applications; e.g., bin packing, cutting stock, layout, loading, knapsack, nesting, orthogonal packing, partitioning, strip packing, trim loss problem and so on.

Although cutting and packing problems have been studied from a long time ago, Gilmore and Gomory's articles in 1960's [42, 43, 44] on linear programming approaches to

one, two and more dimensional cutting stock problems were the first to present techniques which could be practically applied to difficult real-world problems. Since then, articles have been published every year on various types cutting and packing problems. In [104], we can see the numbers of published papers from 1940 to 1989 and the titles of these papers. See Figure 1.1 for a summary of the numbers of published papers. Although

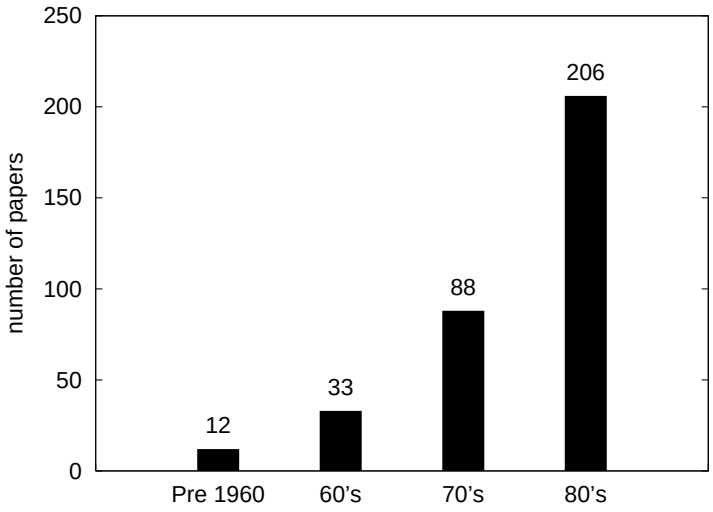


Figure 1.1: Number of published papers between 1940 and 1989

accurate data are not available, more papers have been published in 1990's and 2000's. There are also many survey papers on cutting and packing problems; e.g., Dowsland and Dowsland [28], Dyckhoff [31], Dyckhoff and Finke [32], Hopper and Turton [59], Lodi et al. [78, 80]. From these, we could observe that cutting and packing problems have been an attractive research area for several decades.

The cutting and packing problems can be classified according to their dimensions. The most usual dimensions are 1, 2 and 3, and problems in these categories are more carefully described in the following subsections. It might seem that there is no application for the n-dimensional problem with $n > 3$; however, there are several applications for such problems. For example, the problem to place boxes in a container (three dimensional area) for a fixed period of time can be formulated as a four-dimensional problem. A very special type of the multi-dimensional problem is the so-called the vector packing problem [38], which will be discussed in Subsection 1.2.2.

In the following subsections, we first describe one-dimensional problem, then explain two-dimensional problem, and briefly survey three-dimensional problem. In this thesis, we propose algorithms for the two-dimensional cutting and packing problems, and hence,

we mainly focus on the two-dimensional problems throughout the thesis. However, it is important to survey problems of different dimensions since various techniques have been utilized commonly for different dimensions. Among them, the one-dimensional case is especially important because it is the most basic category of cutting and packing problems.

1.2.1 One-dimensional problem

One-dimensional problem is the most basic category of cutting and packing problems, and many algorithms have been proposed so far. Many of these algorithms for this category have been extended to two and more dimensions. We have three major problems in this category: knapsack, bin packing and cutting stock problems. Each problem is formally described as follows.

Knapsack problem (KP)

input: A set of items I, a size $a(i) \in Z_+$ (the set of nonnegative integer) and a value $c(i) \in Z_+$ for each $i \in I$, and the knapsack capacity $b \in Z_+$.

output: A subset $I' \subseteq I$ of the maximum total value $\sum_{i \in I'} c(i)$ such that the total size $\sum_{i \in I'} a(i)$ is b or less.

Bin packing problem (BPP)

input: A set of items I, a size $a(i) \in Z_+$ for each $i \in I$ and the bin capacity $b \in Z_+$.

output: A partition of I into the minimum number k of disjoint subsets $I_1, I_2, \ldots, I_k$ such that the total size $\sum_{i \in I_j} a(i)$ is b or less for each subset I_j.

Cutting stock problem (CSP)

input: A set of items I, a size $l(i) \in Z_+$ and a demand $d(i) \in Z_+$ for each $i \in I$, and the bin capacity $b \in Z_+$.

output: A family of cutting patterns of items $p_1, p_2, \ldots, p_k$ and the number of applications $x(j)$ for each pattern p_j with the minimum total applications $\sum_j x(j)$ such that all demands are satisfied. Here, a cutting pattern p_j is described as $p_j = (a_{1j}, a_{2j}, \ldots, a_{nj})$, where $a_{ij} \in Z_+$ is the number of item i in pattern p_j, and $\sum_i a_{ij} l(i)$ must be b or less.

Knapsack problem, KP in short, is one of the simplest problems in this category. There are pseudo-polynomial time algorithms and fully polynomial time approximation schemes

for KP [62]. Furthermore, dynamic programming and branch-and-bound algorithms for KP can solve relatively large instances arising in real applications exactly in practical time.

On the other hand, bin packing problem (BPP) is strongly NP-hard and no pseudo-polynomial time algorithm exists unless P = NP. Many researchers have proposed heuristic and approximate algorithms, and the following three methods are basic and used in practice: *next fit, first fit* and *best fit* [69]. For these methods, we first specify a permutation of items i, and then put items into bins in this order based on specific rules.

Algorithm: Next Fit (NF)

Step 1: Set $i := 1, j := 1$ and $\tilde{b}(1) := 0$.

Step 2: If $a(i) + \tilde{b}(j) \leq b$, set $\tilde{b}(j) := \tilde{b}(j) + a(i)$ (i.e., we put item i into the jth bin). Otherwise, set $j := j + 1$ and $\tilde{b}(j) := a(i)$ (i.e., we close the jth bin and open the next bin to place item i).

Step 3: If $i < n$, set $i := i + 1$ and return to Step 2. Otherwise, output j and halt.

Algorithm: First Fit (FF)

Step 1: Set $i := 1, j := 1, k := 1$ and $\tilde{b}(1) := 0$.

Step 2: If $a(i) + \tilde{b}(j) \leq b$, set $\tilde{b}(j) := \tilde{b}(j) + a(i)$ (i.e., we put item i into the jth bin) and go to Step 3. If $a(i) + \tilde{b}(j) > b$ and $j < k$, set $j := j + 1$ and return to Step 2. Otherwise, set $k := k + 1, \tilde{b}(k) := a(i)$ (i.e., we open a new bin and put item i into it), and go to Step 3.

Step 3: If $i < n$, set $i := i + 1, j := 1$ and return to Step 2. Otherwise, output k and halt.

Algorithm: Best Fit (BF)

Step 1: Set $i := 1, j := 1, J := \{1\}$ and $\tilde{b}(1) := 0$.

Step 2: If $a(i) + \tilde{b}(j) > b$ holds for all bins $j \in J$, set $j := j + 1, J := J \cup \{j\}$ and $\tilde{b}(j) := a(i)$ (i.e., we open a new bin and put item i into it). Otherwise, we choose a bin j with the maximum $\tilde{b}(j) \leq b - a(i)$, and set $\tilde{b}(j) := \tilde{b}(j) + a(i)$.

Step 3: If $i < n$, set $i := i + 1$ and return to Step 2. Otherwise, output $|J|$ and halt.

For these methods, the quality of the solution is dependent on the permutation of items. In order to improve the quality of the solution, the following ideas have been developed:

(1) Sorting items with its size in the descending order. Algorithms with this strategy are called *next fit decreasing, first fit decreasing* and *best fit decreasing*, respectively. (2) Generating many permutations randomly and computing solutions for all of the permutations. (3) Searching good permutations of items with local search or metaheuristic algorithms.

Cutting stock problem (CSP) is also a difficult and important problem, which arises in many industries such as steel, paper, wood, glass and fiber. A classical approach to CSP is to formulate it as an integer programming problem (IP), and solve it by a heuristic method based on its linear programming (LP) relaxation. As it is impractical to consider all cutting patterns, which correspond to columns in the LP relaxation, Gilmore and Gomory [42, 43] proposed a column generation technique in 1960's that generates only those columns necessary to improve the lower bound of IP by solving the associated knapsack problems. The LP relaxation often has the property that the rounded up value of the LP lower bound is equal to the optimal value of IP [83]. After that, branch-and-bound, heuristic and metaheuristic algorithms with the column generation technique have been developed with certain computational success [111]. It is however observed that those approaches tend to use many (usually, close to the number of product types $|I|$) different cutting patterns. In recent cutting industries, the setup cost for changing patterns becomes more significant and it is often impractical to use many different cutting patterns. Several researchers (e.g., Foerster and Wäscher [37], Haessler [54], Umetani et al. [108]) have proposed algorithms for CSP with consideration on the number of cutting patterns. In this thesis, we consider a two-dimensional cutting stock problem with consideration on the number of cutting patterns, and propose a heuristic algorithm.

1.2.2 Two-dimensional problem

First, we classify the problems in this category into two types. One is to place items with two independent dimensions (e.g., weight and length) into bins with capacity for each dimension. Another type is to place two-dimensional small items into two-dimensional large objects. The former is the so-called two-dimensional vector packing problem, which is formally described as follows.

Two-dimensional vector packing problem

input: A set of items I, a weight $w(i)$ and a length $l(i)$ for each $i \in I$, and the bin capacity W and L for total weight and length, respectively.

output: A partition of I into the minimum number k of disjoint subsets $I_1, I_2, \ldots, I_k$ such that the total weight $\sum_{i \in I_j} w(i)$ is bounded by W and the total length $\sum_{i \in I_j} l(i)$ is bounded by L for each subset I_j.

There are many studies on this problem and numerous approximate algorithms have been proposed [16, 38]. Note that various techniques for this problem have been extended to the multi-dimensional vector packing problem.

It is not easy to describe the latter type problem (the problem to place two-dimensional small items into two-dimensional large objects) formally since there are still many variations. We first explain how to classify such variations of this problem, and then describe each specific problem formally. The following characteristics are important to classify.

Shapes of small items: (i) rectangle, (ii) polygon, (iii) circle and (iv) irregular object.

Assortment of small items: (i) many different shapes, (ii) many items of relatively few different shapes and (iii) identical shape.

Assortment of large objects: (i) one object, (ii) many objects of identical shape and (iii) many objects with different shapes.

Kind of assignment: (i) the selection of small items to be placed is considered using all available large objects, and (ii) all small items must be placed and the selection of large objects is flexible.

Considering the geometrical nature of the problem, we state the shapes of small items first. The problems with rectangular items may be the most basic and well-studied. In this thesis, we propose some heuristic algorithms for this type of problem. We will describe some specific problems of this type in Chapter 2, together with surveying previous work on the two-dimensional rectangle packing problem.

Now, we briefly review other types of problems (i.e., shapes of small items are not necessarily rectangle). In these decades, there have been many research papers on the two-dimensional polygon packing problem, where several names have been used for the same (or similar) problem, e.g., two-dimensional irregular packing, nesting problem, polygon placement and so on. Also various techniques have been proposed to solve the problem. The most popular techniques used in the previous work may be *bottom left strategy*, *clustering* and *no-fit polygon*. Bottom left strategy was originally proposed for the rectangle

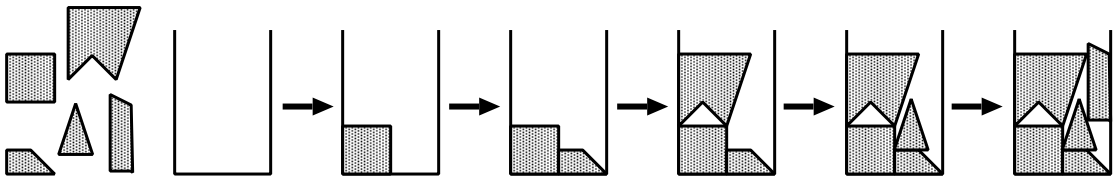


Figure 1.2: Bottom left strategy

packing problem (see Section 2.3 for the detailed explanation), and also applied to the polygon packing problem [29, 51]. Roughly speaking, bottom left strategy places small items one by one at the lowest possible position, left justified. See Figure 1.2 for an example. Clustering is another strategy to place small items. This method repeats the following procedure until all items become one large item: Choose a few items, pack them as compact as possible and consider the resulting combined item as one new item. Figure 1.3 represents an example of this method.

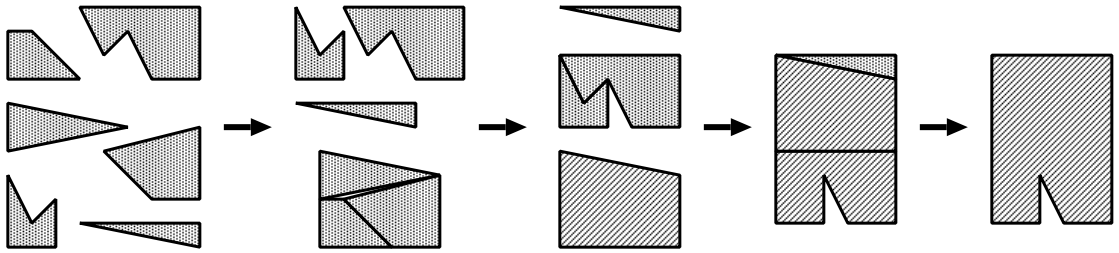


Figure 1.3: Clustering method

If the shape of small items are not convex, it is not trivial to place an item as close as possible to another item without overlap. No-fit polygon [3] is utilized for this purpose. See Figure 1.4 for an example of the no-fit polygon. We compute no-fit polygons for all

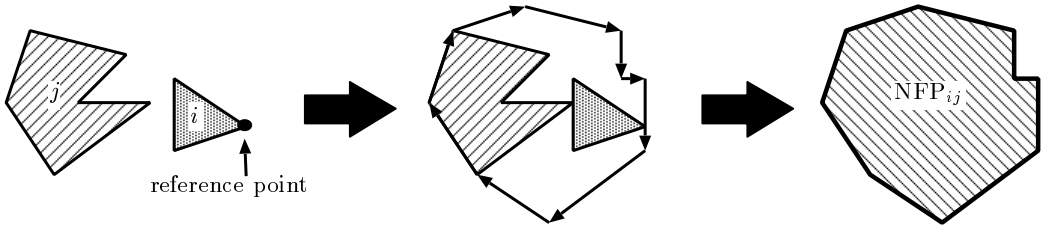


Figure 1.4: No-fit polygon

pairs i and j of items in advance, and place the reference point of item i on the edge of the no-fit polygon NFP_{ij} in order to place i adjacent to j. In many cases, the idea of no-fit polygon is combined with the bottom left strategy and clustering method.

The problem to place circular items has also been studied by many researchers [23, 115]. As we mentioned at the beginning of this section, some problems of this type (usually, we are given identical circles and one large object) have a long history and they have been well studied theoretically. Some other problems (e.g., many different circles are given) have been considered as a result of the rapid progress of computer technology. Figure 1.5

represents an example of this problem where both of small items and large object are circular form. For the circle packing problem, the position of each circle is represented by

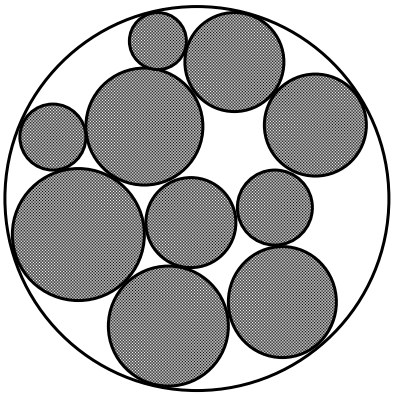


Figure 1.5: Circle packing problem

the coordinates of the center of the circle and its radius, and they are utilized to eliminate overlap.

The problem to pack irregular objects (i.e., each item is of arbitrary shape defined by line segments and/or curves) seems to be very difficult to design effective algorithms. This problem, however, appears in real-world applications, and researchers have proposed various algorithms with some approximation techniques [12, 94] (e.g., each item is described by the line-segment approximation).

1.2.3 Three-dimensional problem

Three-dimensional problem also appears in various real-world applications, e.g., cutting wood or foam rubber into small pieces, loading pallets with goods, and filling containers with cargo. There are several variants of this problem in the literature, e.g., container loading, three-dimensional strip packing, knapsack loading, three-dimensional bin packing, pallet loading and so on, and algorithms have been proposed for those problems [97]. These problems have similar properties to one- and two-dimensional problems, and are more difficult. Hence, most algorithms proposed for the three-dimensional problem have been based on heuristic or metaheuristic algorithms for one- and two-dimensional cutting and packing problems. Note that, there are various original constraints in industrial applications, e.g., increasing stability, support of the load, possibility to carry in (resp., out) items to (resp., from) containers, and we should design algorithms in consideration of each situation. Bischoff and Ratcliff [11] gave an excellent overview of practical requirements which may be imposed to the problem.

1.3 Greedy method, local search and metaheuristics

There are several useful tools to design approximate and heuristic algorithms. We explain three of the most common ideas, namely, greedy method, local search and metaheuristics.

1.3.1 Greedy method

The *greedy method* directly constructs a solution by successively determining the values of variables on the basis of some local information. This method can find optimal solutions for some problems, or find good solutions in many cases for other problems. Here, we show two algorithms which are based on the greedy method. The first example is Kruskal's algorithm [75] for the minimum spanning tree problem.

Kruskal's algorithm

input: A connected undirected graph $G = (V, E)$ and a cost function $c : E \to R$.

output: A spanning tree $H = (V, T)$ (i.e., a connected undirected graph without cycle) with the minimum total cost $\sum_{e \in T} c(e)$.

Step 1: Set $S := E$ and $T := \emptyset$.

Step 2: Find an edge $e \in S$ such that $c(e) \leq c(e')$ for all edges $e' \in S$, and set $S := S \setminus \{e\}$.

Step 3: If there is no cycle in graph $H' = (V, T \cup \{e\})$, set $T := T \cup \{e\}$. If $S = \emptyset$ or $|T| = |V| - 1$ holds, output the minimum cost spanning tree $H = (V, T)$ and stop; otherwise return to Step 2.

This algorithm finds an optimal solution in $O(|E| \log |V|)$ time.

The second example is the nearest neighbor algorithm for the traveling salesman problem (TSP in short, see Section 1.1 for the definition). The nearest neighbor algorithm proceeds as follows:

> Pick any starting vertex. From the current vertex, go to the nearest vertex not visited yet. Repeat this until all vertices have been visited, then return to the starting vertex.

This algorithm runs in $O(n^2)$ time, where n is the number of vertices. Notice that although each move is locally the best possible, the overall result can be quite poor.

1.3.2 Local search

The *local search* is an improvement method that iteratively modifies the current solution to obtain a better solution. The local search, LS in short, starts from an initial feasible

solution $x^{(0)}$ and repeatedly replaces it with a better solution in its *neighborhood* $N(x)$ until no better solution is found in $N(x)$, where $N(x)$ is a set of solutions obtainable from x by applying a slight perturbation. A solution x is *locally optimal*, if no better solution x' is found in $N(x)$. The simple local search algorithm with an initial solution $x^{(0)}$, neighborhood $N(x)$ and the objective function $f(x)$ is formally described as follows.

Algorithm: Local Search (LS)

Step 1: Set $x := x^{(0)}$.

Step 2: If there is a feasible solution $x' \in N(x)$ such that $f(x') < f(x)$ holds, set $x := x'$ and return to Step 2. Otherwise (i.e., $f(x) \leq f(x')$ holds for all solutions $x' \in N(x)$), output the current locally optimal solution x and stop.

The search procedure of finding the next solution x' in Step 2 is called the *neighborhood search*, and the set of all solutions which may be potentially visited in a local search algorithm is called the *search space*. The following ingredients must be specified in designing LS and the performance of an algorithm highly depends on them: Initial solution, neighborhood and move strategy.

1.3.3 Metaheuristics

In general, if only one trial of LS is applied, solutions of better quality may remain unvisited, and it is hard to attain a high quality of the output solution. To overcome this, many variants of simple LS have been developed. We briefly survey their strategies here.

Initial solution: generating a number of different initial solutions to which LS is applied, e.g., multi-start local search (MLS) [70], iterated local search (ILS) [68], greedy randomized adaptive search procedure (GRASP) [36], variable neighborhood search [87] and scatter search [47].

Neighborhood: adopting a sophisticated or larger neighborhood, e.g., variable depth search [70], very large-scale neighborhood search [7] and ejection chain [48, 118].

Move strategy: allowing to move to worse solutions, and controlling the moves by a randomized or sophisticated strategy, e.g., simulated annealing (SA) [71], threshold accepting [30] and tabu search (TS) [46, 50].

Search space: adopting a search space different from the feasible region F (i.e., permitting to search in the infeasible region), and modifying the objective function f so that we can evaluate the amount of infeasibility of solutions. This approach is often

used in approximate algorithms for two reasons: (1) just finding a feasible solution is not easy for many combinatorial optimization problems and (2) many good solutions exist around the boundary between the feasible and infeasible regions.

Evaluation function: adaptively perturbating the evaluation function in order to escape from poor locally optimal solutions, e.g., guided local search [114] and search space smoothing method [52].

The framework of these variants of LS are called *metaheuristics*. In the rest of this subsection, we show some representative metaheuristic algorithms and some algorithms which we will use in this thesis. For more details about local search and metaheuristics, see references [1, 95].

The multi-start local search (MLS) is one of the simplest metaheuristic algorithms. In MLS, we generate many initial solutions randomly, and apply LS to each initial solution independently. Then, we output the best of the obtained locally optimal solutions.

Algorithm: Multi-start Local Search (MLS)

Step 1: Initialize x^* to be an arbitrary solution.

Step 2: Generate a solution x randomly and improve it by LS.

Step 3: If $f(x) \leq f(x^*)$ holds, set $x^* := x$. If some stopping criterion is satisfied, output the best solution x^* found in the search and halt; otherwise return to Step 2.

Here, f is the given objective function. As a stopping criterion, one of the following three criteria is usually used: (1) computational time, (2) number of iterations and (3) number of consecutive "failure" iterations; i.e., if x^* is not improved for the predetermined number of consecutive iterations, output the best solution x^* and stop.

The iterated local search (ILS) is a variant of MLS, in which initial solutions are generated by slightly perturbing a good solution x_{seed} obtained so far. It is important to generate initial solutions which retain some features of solution x_{seed} and to avoid a cycling of solutions in order to improve the performance of ILS. Here, we describe an iterated local search algorithm which uses the best obtained solution x^* as x_{seed}.

Algorithm: Iterated Local Search (ILS)

Step 1: Initialize x^* to be an arbitrary solution.

Step 2: Generate a solution x by slightly perturbing x^* randomly.

Step 3: Improve x by LS.

Step 4: If $f(x) \leq f(x^*)$ holds, set $x^* := x$. If some stopping criterion is satisfied, output x^* and halt; otherwise return to Step 2.

The genetic algorithm (GA) [57] is based on the evolutionary process in nature. GA repeatedly generates a set of new solutions Q by applying the operations *crossover* and/or *mutation* to the set of current solutions P. A crossover generates one or more new solutions by combining two or more current solutions, and a mutation generates a new solution by slightly perturbating a current solution. GA starts from an initial set of solutions P and repeatedly replaces P with $P' \subseteq P \cup Q$ according to its selection rule.

Algorithm: Genetic Algorithm (GA)

Step 1: Generate an initial set of solutions P and let x^* be the best solution among P.

Step 2: Repeat the following steps 2-1 and 2-2 until the set of new solutions Q are obtained where the cardinality of Q is prespecified.

2-1: Choose two or more solutions belong to P and crossover them to generate one or more new solutions.

2-2: Choose a solution belongs to P and mutate it to generate a new solution.

Step 3: If there is a solution $x \in Q$ with $f(x) < f(x^*)$, choose a best solution $x \in Q$ and set $x^* := x$.

Step 4: Select a set of solutions P' (of a prespecified size) from the resulting $P \cup Q$, and set $P := P'$.

Step 5: If some stopping criterion is satisfied, output the best obtained solution x^* and halt; otherwise return to Step 2.

In Step 4, the following strategies to make a new set of solutions P' are often used: random selection, roulette wheel selection, and elitism. The above framework is called as the simple genetic algorithm, and it is known that the quality of its solution is not as good as other metaheuristics for most problems. Brady [13] combined GA with LS, which is called the genetic local search (GLS), and it attained certain computational success. For more details about the genetic algorithm, see reference [25].

The simulated annealing (SA) is a kind of probabilistic local search, in which test solutions are randomly chosen from $N(x)$ and accepted with probability that is 1 if the test solution is better than the current solution x, and is positive even if it is worse than x. By allowing moves to worse solutions, SA is able to escape from poor locally

optimal solutions. The acceptance probability of moves is controlled by a parameter called *temperature*, whose idea stems from the physical process of annealing.

Algorithm: Simulated Annealing (SA)

Step 1: Generate an initial solution x randomly and set $x^* := x$. Determine the initial temperature t.

Step 2: Generate a solution $x' \in N(x)$ randomly, and set $\Delta := f(x') - f(x)$. If $\Delta < 0$ holds (i.e., a better solution is found), set $x := x'$; otherwise set $x := x'$ with probability $e^{-\Delta/t}$.

Step 3: If $f(x) < f(x^*)$ holds, set $x^* := x$. If some stopping criterion is satisfied, output x^* and halt; otherwise update the temperature t according to some rule and return to Step 2.

A rule to adjust the temperature is called cooling schedule, and various ideas were proposed [22]. One of the simplest rules is the geometric cooling, where we update $t := \alpha t$ ($0 < \alpha < 1$ is a parameter) at intervals of the prespecified iterations.

The tabu search (TS) tries to enhance LS by using the memory of previous searches. TS repeatedly replaces the current solution x with its best neighbor $x' \in N(x) \setminus (\{x\} \cup T)$ even if $f(x') \geq f(x)$ holds, where the set T, called the *tabu list*, is a set of solutions which includes those solutions most recently visited. Cycling of a short period can be avoided as a result of introducing tabu list. See reference [50] for detailed explanation of the tabu search.

Algorithm: Tabu Search (TS)

Step 1: Generate an initial solution x. Set $x^* := x$ and $T := \emptyset$.

Step 2: Find the best solution $x' \in N(x) \setminus (\{x\} \cup T)$, and set $x := x'$.

Step 3: If $f(x) < f(x^*)$ holds, set $x^* := x$. If some stopping criterion is satisfied, output the best obtained solution x^* and halt; otherwise update T according to some rule and return to Step 2.

Theoretically, very little is known about nontrivial bounds on the quality and the time complexity of local search. There are a few theoretical results for metaheuristic algorithms, e.g., asymptotic convergence of SA under some appropriate assumptions [40, 82], similar results for GA and the probabilistic tabu search [33, 101], and finite time convergence of TS [49, 55]; however, these results do not assure that we can find good solutions in realistic time.

However, in practice, many local search and metaheuristic algorithms are successful to obtain sufficiently good solutions in reasonable computational time. One of the attractive features of local search and metaheuristics consists in its simplicity and robustness. We can develop local search and metaheuristic algorithms without knowing detailed mathematical properties of the problem, and still attain reasonably good solutions in practically feasible time [117]. Another good feature comes from its flexibility, i.e., we can further improve their performance by introducing sophisticated data structures and effective heuristics of the problem.

1.4 Research objectives and overview of the thesis

As we have seen in this chapter, cutting and packing problems have the following properties:

- intractable (NP-hard) to compute exact optimal solutions,
- important in real-world applications,
- there are many variations of the problems.

Therefore, practical approximate algorithms for cutting and packing problems are needed; in fact many heuristic and metaheuristic algorithms have been proposed during these several decades. However still people in application fields cannot solve their specific problems in the practical sense, nor afford to invest sufficient manpower and time to develop individual algorithms. Our purpose in this thesis is to introduce a new problem, to which a wide range of specific problems can be reducible, and develop practical approximate algorithms for this problem. In general, however, when we consider a general problem encompassing many problems, problem structures of individual problems cannot be exploited, and hence it becomes hard to attain higher performance. We therefore restrict the target of this study to the two-dimensional rectangle packing problem, and consider a general framework of this problem. Note that, there are still many variants for the rectangle packing problem, and we show some of them in Chapter 2. We would like to attain as high performance as specialized algorithms for various types of the rectangle packing problem.

The thesis is organized as follows. In Chapter 2, we define various specific problems of the two-dimensional rectangle packing problem (2DRPP). We survey previous work on 2DRPP and explain several basic techniques for solving the problem. We then propose a new general formulation of the rectangle packing problem. The problem we consider in this thesis is the two-dimensional rectangle packing problem with general spatial costs, which is characterized by its spatial cost functions and the modes for each rectangle. In Chapter 2, we show that numerous problems in practice (not only various types of the

rectangle packing problems but also some scheduling problems) can be reduced to our problem.

In Chapter 3, we propose local search algorithms for the rectangle packing problem with general spatial costs (RPGSC). In order to design algorithms for RPGSC, we must specify the following ingredients.

1. Coding scheme: How to represent a solution.
2. Decoding algorithm: How to compute a placement from a coded solution.
3. Search strategy: How to find a good coded solution.

To represent a solution of this problem, we utilize the sequence pair coding scheme which was proposed by Murata et al. [90]. We propose two different decoding algorithms to find an optimal location of the rectangles using dynamic programming. In order to find a good coded solution, we propose metaheuristic algorithms based on local search. To make the search efficient, we define the critical path that corresponds to the bottleneck of the current solution, and make use of it to reduce the sizes of various neighborhoods. The local search algorithms based on these neighborhoods are then incorporated in metaheuristic algorithms. We report computational results on various implementations, and compare our algorithms with other existing algorithms for various types of the rectangle packing problem and a real-world scheduling problem.

In Chapter 4, we tackle the same problem RPGSC of Chapter 3. In this chapter, we propose new decoding algorithms which improve the amortized time complexity of the previous decoding algorithms. The new decoding algorithms work efficiently in the neighborhood search. We then design metaheuristic algorithms with the new decoding algorithms. The computational results for the rectangle packing problem and a real-world scheduling problem are reported, and they exhibit good prospects of the proposed algorithms.

In Chapter 5, we consider a variant of the two-dimensional rectangle packing problem, called the two-dimensional cutting stock problem with a given number of different patterns. As we mentioned in Subsection 1.2.1, the setup cost for changing patterns is more dominant in recent manufacturing industries, and it is impractical to use many different cutting patterns. This motivated us to reduce both of the number of different cutting patterns and the unused area. In our local search algorithms, we use several heuristic algorithms (including the rectangle packing algorithms in Chapters 3 and 4) as their components. We generate random test instances of this problem and conduct computational experiments to compare our algorithms with various different neighborhood operations. We also compute the trade-off curves between the number of different cutting patterns m and the solution quality.

Finally, in Chapter 6, we summarize our study in this thesis.

Chapter 2

Two-Dimensional Rectangle Packing Problem

2.1 Introduction

In this chapter, we give the definitions of the two-dimensional rectangle packing problem (2DRPP) and introduce some specific problems with various constraints. For 2DRPP, there are many coding schemes and decoding methods in the literature. We briefly review them and explain heuristic and metaheuristic algorithms with these schemes and methods. Then, we propose a new formulation of 2DRPP with general spatial costs and modes for each rectangle. Defining spatial cost functions and modes appropriately, various cutting and packing problems and scheduling problems can be formulated in this form. We show some examples of such problems.

2.2 Formulations of 2DRPP

The two-dimensional rectangle packing problem, 2DRPP in short, is defined as follows. We are given n small rectangles $I = \{1, 2, \ldots, n\}$, where each rectangle $i \in I$ has a width w_i and a height h_i, and two-dimensional large objects with rectangular form. We place small rectangles into large objects to minimize a given objective function.

As we mentioned in the previous chapter, there are still many variations for this problem, and the following characteristics are important to classify: kind of assignment, assortment of large objects, assortment of small rectangles, and orientation of rectangles. We will review some specific variations of the two-dimensional rectangle packing problem, and give various formulations for those in this section.

First, if a selection of small rectangles are placed in one large object of given size,

the problem is called the two-dimensional knapsack problem, and one of the most basic formulations of this type is as follows.

Two-dimensional knapsack problem

input: A set of small rectangles $i \in I$ with its width w_i, height h_i and value c_i, and a two-dimensional knapsack with its width W and height H.

output: A subset $I' \subseteq I$ of the small rectangles with maximum total value $\sum_{i \in I'} c_i$ such that all rectangles $i \in I'$ can be placed in the knapsack without mutual overlap.

There are several variants of this problem, e.g., each small rectangle i can be selected up to b_i times, where b_i is a given upper bound. There have been many studies on these problems and numerous algorithms were proposed [24, 81, 109, 116].

We then consider several types of the rectangle packing problem with a focus on the large rectangular objects. We explain three different types of problems called the area minimization, strip packing and two-dimensional bin packing, respectively. For each problem, we are given a set of small rectangles i with its width w_i and height h_i.

Area minimization problem: We are given one large object whose width and height are decision variables. The objective is to minimize the area of the large rectangular object that covers all of the small rectangles [90, 92, 105, 106].

Strip packing problem: We are given one large object (called strip) whose width W is given and height H is a decision variable. The objective is to minimize the height H of the strip subject to all small rectangles are placed in the strip without mutual overlap [58, 77, 85].

Two-dimensional bin packing problem: We are given an unlimited number of large objects (rectangular bins), having identical width W and height H. The objective is to minimize the number of bins utilized to place all small rectangles [10, 79, 80].

We then mention the following two problems, called the two-dimensional cutting stock and pallet loading, which have distinct characters on the small rectangles. We first consider the two-dimensional cutting stock problem. We define several terms before going to the definition of the problem. A cutting pattern p_j is described as $p_j = (a_{1j}, a_{2j}, \ldots, a_{nj})$, where $a_{ij} \in \boldsymbol{Z}_+$ is the number of small rectangle i (called product) cut from a large object (called stock sheet) by pattern p_j. A placement of products in a pattern is a set of their locations in one stock sheet, where a placement is feasible if all the products are placed in

one stock sheet without mutual overlap. We call a pattern p_j feasible if it has a feasible placement. Let S denote the set of all feasible patterns. Note that, the set S is very large and it is not explicitly given; i.e., we must find a feasible placement to confirm that a pattern is feasible. Now, the two-dimensional cutting stock problem is formally described as follows.

Two-dimensional cutting stock problem

input: A set of small rectangles $i \in I$ with its width w_i, height h_i and demand d_i, and an unlimited number of large objects (rectangular bins), having identical width W and height H.

output: A set of cutting patterns $\Pi = \{p_1, p_2, \ldots, p_{|\Pi|}\} \subseteq S$, and the numbers of applications $X = (x_1, x_2, \ldots, x_{|\Pi|})$ of all the patterns $p_j \in \Pi$, where S is the set of feasible cutting patterns and $x_j \in \boldsymbol{Z}_+$. The objective is to minimize the total number of stock sheets $\sum_{p_j \in \Pi} x_j$ subject to the demand constraint $\sum_{p_j \in \Pi} a_{ij} x_j \geq d_i$ for each rectangle $i \in I$.

This is an important problem for various real-world applications such as the manufacturing industry. In 1965, Gilmore and Gomory extended their work on the (one-dimensional) cutting stock problem [42, 43] to this problem [44]. They proposed a column generation scheme in which new cutting patterns are produced by solving the generalized knapsack problems. After this paper, many heuristic algorithms based on the column generation technique have been proposed [18, 24, 34, 109, 110, 112]. In this thesis, we consider a variant of this problem considering the number of different cutting patterns $|\Pi|$. It is called the two-dimensional cutting stock problem with a given number of different patterns, and we propose heuristic algorithms for this problem with an aim of minimizing both of the total number of stock sheets and the number of different cutting patterns. The details will be described in Chapter 5.

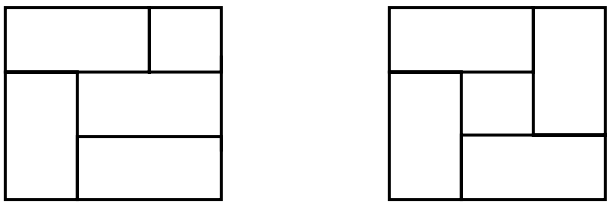
Now, we define another type of the rectangle packing problem called the pallet loading problem. This problem has also been studied by many researchers [76, 88]. The complexity of this problem is one of the well-known open problems in this field. That is, no polynomial time algorithm to find an optimal solution nor the verification of the NP-hardness of this problem is known. The problem is formally defined as follows.

Pallet loading problem

input: A small rectangle with its width w and height h, and a large rectangular object with its width W and height H.

output: The maximum number of small rectangles which are placed on the large object under the condition that no two rectangles overlap, where each rectangle can be rotated by 90°.

Finally, we explain some other constraints for the rectangle packing problem. The rectangle packing problem is closely related to the real-world applications, and numerous original constraints for each specific problem arise. Among them, orientation and guillotine cut constraint are especially important. For some problems, we assume that the small rectangles have a fixed orientation (i.e., they cannot be rotated), and rotation (usually by 90°, sometimes arbitrary) may be allowed for other problems. Guillotine cut constraint signifies that the small rectangles must be obtained through a sequence of edge-to-edge cuts parallel to the edges of the large object (see Figure 2.1 as an example), which is usually imposed by technical limitations of the automated cutting machines.



(a) guillotine cut (b) non-guillotine cut

Figure 2.1: Guillotine cut constraint

Several other constraints on the rectangle packing problem can be found in the literature and real-world applications; e.g., the maximum number of small rectangles for one large object is limited, the location of some small rectangles are fixed and so on.

2.3 Previous work on 2DRPP

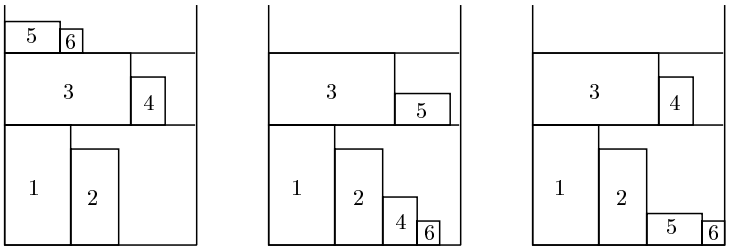
In this section, we will review previous work on 2DRPP. If we search directly the x and y coordinates of each small rectangle, an effective search will be difficult, since the number of solutions is uncountably many and eliminating overlap of rectangles is not easy. Therefore numerous coding schemes to represent solutions have been proposed [8, 19, 21, 58, 77, 79, 90, 92, 105, 116]. We review various algorithms for 2DRPP with a focus on the coding schemes and decoding algorithms.

One of the most popular coding schemes is to represent a solution by a permutation of n rectangles. Various decoding algorithms have been proposed for this coding scheme;

they place rectangles one by one in this order based on some rule. Most of the approaches are *level algorithms*, i.e., placement is obtained by placing rectangles, from left to right, in rows forming levels. The first level is the bottom of the large object, and subsequent levels are produced by the horizontal line coinciding with the top of the tallest rectangle packed on the level below. As we mentioned in Subsection 1.2.1, the next fit, first fit and best fit are famous methods for the one-dimensional problem, and these methods have been extended to the two-dimensional rectangle packing problem. In each case, the small rectangles are initially sorted in the non-increasing order of height, and packed in this order. Let i denote the current rectangle, and s the level created most recently.

- *Next fit decreasing height* (NFDH) strategy: rectangle i is packed left justified on level s, if it fits. Otherwise, a new level ($s := s + 1$) is created and i is packed left justified into it.
- *First fit decreasing height* (FFDH) strategy: rectangle i is packed left justified on the first level where it fits, if any. If no level can accommodate i, a new level is initialized as in NFDH.
- *Best fit decreasing height* (BFDH) strategy: rectangle i is packed left justified on the level that minimizes the unused horizontal space among those where it fits. If no level can accommodate i, a new level is initialized as in NFDH.

The above strategies are illustrated through an example in Figure 2.2.



(a) NFDH (b) FFDH (C) BFDH

Figure 2.2: Three classical level algorithms

There are many theoretical and computational results for these algorithms. For example, Coffman et al. [21] analyzed NFDH and FFDH for the solution of the strip packing problem and determined their asymptotic worst-case behavior:

$$\mathrm{NFDH}(\mathcal{I}) \leq 2 \cdot \mathrm{OPT}(\mathcal{I}) + 1,$$
$$\mathrm{FFDH}(\mathcal{I}) \leq \frac{17}{10} \cdot \mathrm{OPT}(\mathcal{I}) + 1,$$

where $\mathcal{I}$ is an instance of the strip packing problem such that the heights of rectangles are normalized as $\max_i h_i = 1$, NFDH($\mathcal{I}$) (resp., FFDH($\mathcal{I}$)) is the height of the strip (objective value) by algorithm NFDH (resp., FFDH) for instance $\mathcal{I}$, and OPT($\mathcal{I}$) is the optimal height of the strip (optimal value) for instance $\mathcal{I}$.

Chung et al. [20] proposed a two-phase algorithm for the two-dimensional bin packing problem called the hybrid first fit, and analyzed its approximation ratio. This algorithm is briefly described as follows (see Figure 2.3 as an example): We first pack rectangles into

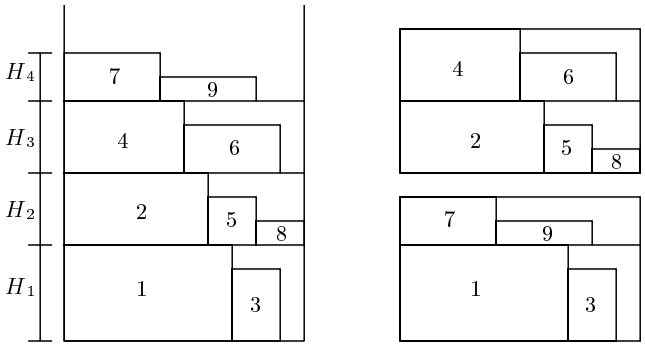


Figure 2.3: Hybrid first fit strategy for the two-dimensional bin packing problem

one strip of width W by FFDH, and set H_i as the height of each level. Then solve the one-dimensional bin packing problem with item sizes H_i and bin capacity H through the first fit algorithm for BPP.

A different classical approach, which does not pack the rectangles by levels, was proposed by Baker et al. [8]. Their algorithm is called the *bottom left* (BL) algorithm: We first specify a permutation of the given rectangles and then pack them one by one in this order at the lowest possible position, left justified. This approach is illustrated through an example in Figure 2.4. There are many variants of this algorithm [19, 58, 77], and it seems that this strategy works well for many specific problems.

There are still many algorithms which utilize the permutation to represent a solution. For example, Lodi et al. [79] proposed several decoding algorithms such as *floor ceiling*, *alternate directions* and *touching perimeter*, and computationally compared with other classical decoding algorithms. Wu et al. [116] proposed a complicated decoding algorithm called as effective quasi-human based heuristic, and achieved certain computational success.

Now, we explain other coding schemes for the rectangle packing problem. The schemes we explain here describe the relative locations for each pair of rectangles by a coded

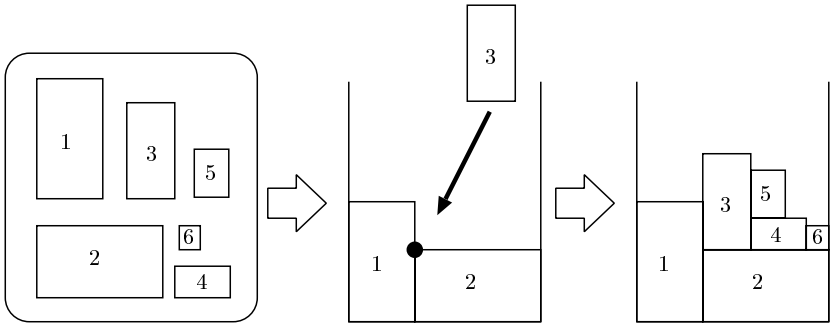


Figure 2.4: BL algorithm for the strip packing problem

solution, and decoding algorithms for those coding schemes solve the two-dimensional rectangle packing problem under the given constraints (i.e., relative locations).

Nakatake et al. [92] proposed a coding scheme called the bounded-sliceline grid (BSG). BSG (see Figure 2.5 as an example) consists of a set of small rooms which are separated by horizontal and vertical segments. To represent a solution, we put all the rectangles

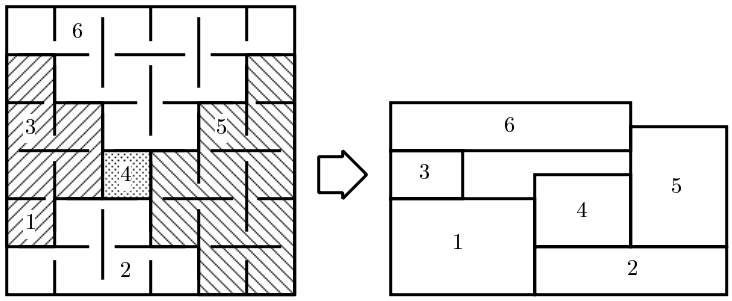


Figure 2.5: Bounded slice-line grid

into rooms, where at most one rectangle can be placed in each room. Based on this coded solution, we assign relative locations for each pair of rectangles, e.g., in Figure 2.5, "4 is right of 1", "4 is above 2", "4 is left of 5" and so on. Nakatake et al. proposed a decoding algorithm (i.e., algorithm to compute an optimal placement under the constraints on relative locations) which runs in linear time of the number of small rooms. They also proposed a simulated annealing algorithm based on BSG, and obtained good computational results for the area minimization problem.

Murata et al. [90] proposed a coding scheme called the sequence pair. For the sequence pair representation, a solution is represented by a pair of permutations of n rectangles (see Figure 2.6 as an example). Based on this coded solution, we assign relative locations for

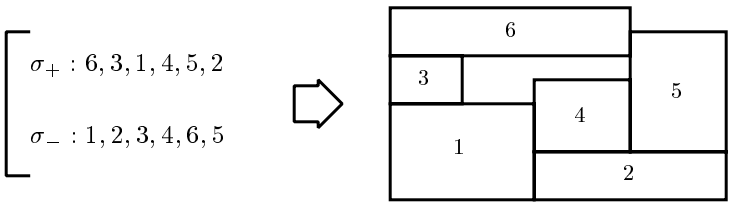


Figure 2.6: Sequence pair

each pair i and j of rectangles as follows: If i is before j in both permutations, we must place i left of j. If i is before j in σ_+ and after j in σ_-, we place i above j. For example, in Figure 2.6, "1 is before 4 in both permutations, and hence 1 is left of 4", "4 is before 2 in σ_+ and after 2 in σ_-, and hence 4 is above 2" and so on.

In Chapters 3 and 4, we propose heuristic algorithms for RPGSC with this coding scheme. We will give a more precise definition of the sequence pair in Section 3.2. For this coding scheme, Murata et al. [90] proposed an $O(n^2)$ time decoding algorithm to obtain an optimal placement (i.e., x and y coordinates of rectangles) from a pair of permutations of rectangles. Takahashi [105] and Tang et al. [106] improved the time complexity of the decoding algorithm to $O(n \log n)$; Tang and Wong [107] further improved it to $O(n \log \log n)$.

There are other coding schemes which describe the relative locations for each pair of rectangles. For example, Guo et al. [53] proposed an ordered tree (called O tree) structure to represent a solution, and Sakanushi et al. [102] proposed a coding scheme called the quarter-state-sequence.

2.4 2DRPP with general spatial costs (RPGSC)

In this section, we propose the rectangle packing problem with general spatial costs, RPGSC in short. This problem is very general, and various types of rectangle packing problems and scheduling problems can be formulated in this form. We first formulate the problem formally, and then show several problems which are reducible to RPGSC.

2.4.1 Formulation

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,

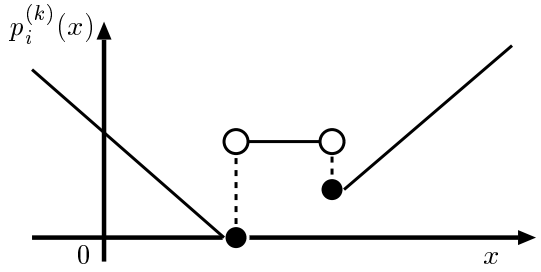


Figure 2.7: An example of the spatial cost function

- 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 these functions are lower semi-continuous; that is, any discontinuous point x (if exists) must satisfy

$$p_i^{(k)}(x) \leq \lim_{\varepsilon \to 0} \min\{p_i^{(k)}(x+\varepsilon), p_i^{(k)}(x-\varepsilon)\} \tag{2.4.1}$$

(see Figure 2.7 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 thesis (unless otherwise stated) that the information of each linear piece of functions $p_i^{(k)}(x)$ and $q_i^{(k)}(y)$ are given explicitly, and each function is represented by the linked lists (see Figure 2.8 as an example). 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

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

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:

$$p_{\max}(\pi) = \max_{i \in I} p_i^{(\mu_i(\pi))}(x_i(\pi)), \tag{2.4.2}$$

$$q_{\max}(\pi) = \max_{i \in I} q_i^{(\mu_i(\pi))}(y_i(\pi)). \tag{2.4.3}$$

Now we are given two cost functions $g(p_{\max}(\pi), q_{\max}(\pi))$ and $c(\mu(\pi))$. We assume that function g is nondecreasing in $p_{\max}(\pi)$ and $q_{\max}(\pi)$, and can be computed in $O(1)$ time

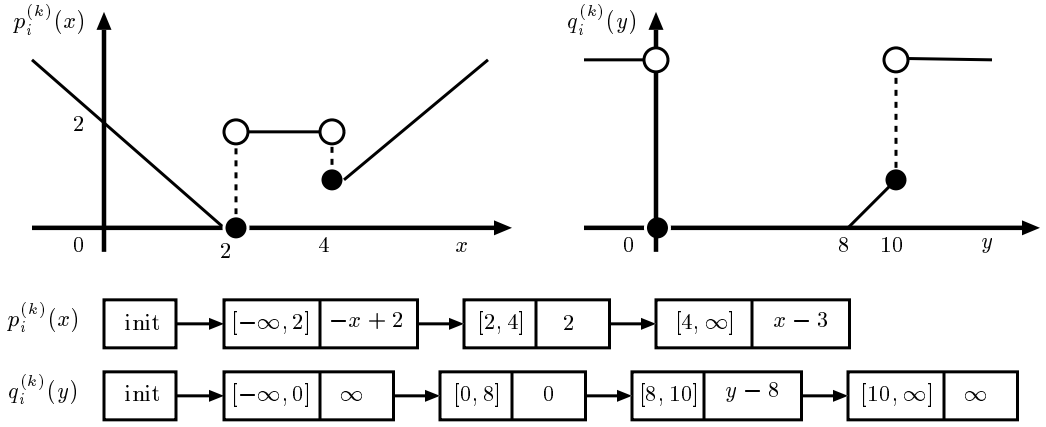


Figure 2.8: Linked lists representing piecewise linear functions

for given $p_{\max}(\pi)$ and $q_{\max}(\pi)$. Moreover, we assume that $c(\mu(\pi))$ can be computed in $O(n)$ time for a given $\mu(\pi)$, and $c(\mu(\pi)) - c(\mu(\pi'))$ can be computed in $O(|\{i \in I \,|\, \mu_i(\pi) \neq \mu_i(\pi')\}|)$ time[1] for given $\mu(\pi)$ and $\mu(\pi')$. Then, the rectangle packing problem with general spatial costs 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:

$$x_i(\pi) + w_i^{(\mu_i(\pi))} \leq x_j(\pi), \tag{2.4.4}$$

$$x_j(\pi) + w_j^{(\mu_j(\pi))} \leq x_i(\pi), \tag{2.4.5}$$

$$y_i(\pi) + h_i^{(\mu_i(\pi))} \leq y_j(\pi), \tag{2.4.6}$$

$$y_j(\pi) + h_j^{(\mu_j(\pi))} \leq y_i(\pi). \tag{2.4.7}$$

The constraints from (2.4.4) to (2.4.7) mean that no two rectangles overlap in packing π, and we call a packing satisfying all constraints feasible. For example, condition (2.4.4) means that the right side of rectangle i is placed to the left of the left side of rectangle j. Condition (2.4.6) means that the upper hem of rectangle i is placed below the lower hem of rectangle j.

[1]We assume that we have already computed $c(\mu(\pi'))$, stored useful information and could utilize it in the computation of $c(\mu(\pi)) - c(\mu(\pi'))$ if necessary. This assumption is natural in local search and metaheuristic algorithms.

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 cutting and packing problems and scheduling problems can be formulated in our form. For example, we can treat a rectangle packing problem 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 will show some specific problems which can be formulated as RPGSC in the following subsection.

2.4.2 Problems reducible to RPGSC

In this section, we show several problems which are reducible to RPGSC. Some of the problems have already been defined in Chapters 1 and 2, but we will describe them again for readability. The first example is the (one-dimensional) bin packing problem (defined in Subsection 1.2.1), which is known to be strongly NP-hard [39].

Bin packing problem

input: A set of items I, a size $a(i) \in Z_+$ for each $i \in I$ and the bin capacity $b \in Z_+$.

output: A partition of I into the minimum number k of disjoint subsets $I_1, I_2, \ldots, I_k$ such that the total size $\sum_{i \in I_j} a(i)$ is b or less for each subset I_j.

This problem is polynomially reducible to RPGSC by the following reduction. In the resulting RPGSC instance, each rectangle has only one mode (i.e., $m_i = 1$ for all i). For each $i = 1, 2, \ldots, n$, we set

$$w_i^{(1)} = a(i), \qquad h_i^{(1)} = 1,$$
$$p_i^{(1)}(x) = \begin{cases} +\infty & (x < 0) \\ 0 & (0 \le x \le b - w_i^{(1)}) \\ +\infty & (x > b - w_i^{(1)}), \end{cases} \qquad q_i^{(1)}(y) = \begin{cases} +\infty & (y < 0) \\ y + 1 & (y \ge 0). \end{cases}$$

Here, $p_{\max}(\pi) = +\infty$ means that there is at least one subset I_j such that the total size $\sum_{i \in I_j} a(i)$ is more than b, and $q_{\max}(\pi)$ denotes the number of disjoint subsets k. The objective of the reduced RPGSC instance is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$.

Various types of the two-dimensional rectangle packing problems explained in this chapter can be also transformed to RPGSC. The first example is the area minimization problem with fixed orientation.

Area minimization problem

input: The set of n rectangles $i \in I$ with its width w_i and height h_i.

output: A packing π such that all rectangles in I are packed in a rectangular object without overlap and the area of the object is minimized, where the orientation of each small rectangle is fixed.

This problem is polynomially reducible to RPGSC by the following reduction. In the reduced RPGSC instance, each rectangle has only one mode. For each $i = 1, 2, \ldots, n$, we set

$$w_i^{(1)} = w_i, \qquad h_i^{(1)} = h_i,$$
$$p_i^{(1)}(x) = \begin{cases} +\infty & (x < 0) \\ x + w_i^{(1)} & (x \geq 0), \end{cases} \qquad q_i^{(1)}(y) = \begin{cases} +\infty & (y < 0) \\ y + h_i^{(1)} & (y \geq 0). \end{cases}$$

$p_{\max}(\pi)$ (resp., $q_{\max}(\pi)$) denotes the width (resp., the height) of the large rectangular object, and the objective of the reduced RPGSC is to minimize $p_{\max}(\pi) \cdot q_{\max}(\pi)$ (i.e., the area of the large object).

The next problem is the strip packing problem in which small rectangles can be rotated by 90°. This problem is formally described as follows.

Strip packing problem

input: The set of n rectangles $i \in I$ with its width w_i and height h_i, and the width W of the large object called strip.

output: The minimum height H of the strip such that all small rectangles are packed in the strip without overlap, where each rectangle can be rotated by 90°.

In the reduced RPGSC instance, each rectangle has two modes corresponding to its orientations: (1) the original orientation and (2) the orientation rotated by 90°. For each $i = 1, 2, \ldots, n$, we set

$$w_i^{(1)} = w_i, \qquad h_i^{(1)} = h_i,$$
$$w_i^{(2)} = h_i, \qquad h_i^{(2)} = w_i.$$

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

$$p_i^{(k)}(x) = \begin{cases} +\infty & (x < 0) \\ 0 & (0 \le x \le W - w_i^{(k)}) \\ +\infty & (x > W - w_i^{(k)}), \end{cases} \qquad q_i^{(k)}(y) = \begin{cases} +\infty & (y < 0) \\ y + h_i^{(k)} & (y \ge 0). \end{cases}$$

Here, $p_{\max}(\pi) = +\infty$ means the packing is infeasible and $q_{\max}(\pi)$ denotes the height of the strip. The objectives of the reduced RPGSC is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$.

The following example is the two-dimensional knapsack problem with rotation.

Two-dimensional knapsack problem

input: A set of small rectangles $i \in I$ with its width w_i, height h_i and value c_i, and the two-dimensional knapsack with its width W and height H.

output: A subset $I' \subseteq I$ of the rectangles with maximum total value $\sum_{i \in I'} c_i$ such that all rectangles $i \in I'$ can be placed in the knapsack without mutual overlap, where each rectangle can be rotated by 90°.

This problem is also polynomially reducible to RPGSC. Each rectangle has the following three modes: (1) $i \in I'$ with the original orientation, (2) $i \in I'$ with the orientation after 90° rotation, and (3) $i \notin I'$. For each $i = 1, 2, \ldots, n$, we set

$$\begin{array}{lll} w_i^{(1)} = w_i, & h_i^{(1)} = h_i, & c_i^{(1)} = 0, \\ w_i^{(2)} = h_i, & h_i^{(2)} = w_i, & c_i^{(2)} = 0, \\ w_i^{(3)} = 0, & h_i^{(3)} = 0, & c_i^{(3)} = c_i. \end{array}$$

For each $i = 1, 2, \ldots, n$ and $k = 1, 2, 3$, we set

$$p_i^{(k)}(x) = \begin{cases} +\infty & (x < 0) \\ 0 & (0 \le x \le W - w_i^{(k)}) \\ +\infty & (x > W - w_i^{(k)}), \end{cases} \qquad q_i^{(k)}(y) = \begin{cases} +\infty & (y < 0) \\ 0 & (0 \le y \le H - h_i^{(k)}) \\ +\infty & (y > H - h_i^{(k)}). \end{cases}$$

$p_{\max}(\pi) + q_{\max}(\pi) = 0$ (resp., $= +\infty$) means that all of the selected rectangles $i \in I'$ are placed in the knapsack (resp., there is at least one rectangle $i \in I'$ which is not placed in the knapsack), and $\sum_{i \in I} c_i^{(\mu_i(\pi))}$ is the total value $\sum_{i \notin I'} c_i$ of the unselected small rectangles. Note that, the maximization of $\sum_{i \in I'} c_i$ is the same as the minimization of $\sum_{i \notin I'} c_i$. Therefore, the objective of the reduced RPGSC instance is to minimize $p_{\max}(\pi) + q_{\max}(\pi) + \sum_{i \in I} c_i^{(\mu_i(\pi))}$.

The last example for the rectangle packing problem is the two-dimensional bin packing problem with rotation.

Two-dimensional bin packing problem

input: A set of n rectangles $i \in I$ with its width w_i and height h_i, and an unlimited number of large objects (rectangular bins), having identical width W and height H.

output: A partition of I into the minimum number k of disjoint subsets $I_1, I_2, \ldots, I_k$ such that all of the small rectangles $i \in I_j$ can be placed in one rectangular object for each subset I_j.

In the reduced RPGSC instance, each rectangle i has two modes corresponding to its orientation. For each $i = 1, 2, \ldots, n$, we set

$$w_i^{(1)} = w_i, \qquad h_i^{(1)} = h_i,$$
$$w_i^{(2)} = h_i, \qquad h_i^{(2)} = w_i.$$

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

$$p_i^{(k)}(x) = \begin{cases} +\infty & (x < 0) \\ 1 & (0 \le x \le W - w_i^{(k)}) \\ +\infty & (W - w_i^{(k)} < x < W) \\ 2 & (W \le x \le 2W - w_i^{(k)}) \\ +\infty & (2W - w_i^{(k)} < x < 2W) \\ & \vdots \\ n & ((n-1)W \le x \le nW - w_i^{(k)}) \\ +\infty & (x > nW - w_i^{(k)}), \end{cases}$$

$$q_i^{(k)}(y) = \begin{cases} +\infty & (y < 0) \\ 0 & (0 \le y \le H - h_i^{(k)}) \\ +\infty & (y > H - h_i^{(k)}). \end{cases}$$

If, $p_i^{(k)}(x)$ is finite, it represents the index of the rectangular bin in which rectangle i is placed and hence $p_{\max}(\pi)$ becomes the number of bins used in the solution. The objective of the reduced RPGSC instance is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$.

These are a part of problems reducible to RPGSC and many other cutting and packing problems can be reduced to our form. As we mentioned before, not only these problems but also some scheduling problems are reducible to RPGSC. Now, we give two examples of scheduling problems, called Scheduling-1 and Scheduling-2, and show that they are special cases of RPGSC.

Scheduling-1 is the problem of scheduling n jobs to m parallel machines. A schedule is determined by an assignment of jobs to machines and the start time of each job. For

each job i, we are given a due date d_i and a processing time t_{ik} if it is processed on machine k. A machine can process at most one job at a time, and the process of a job can not be stopped during its processing time. Let C_i be the completion time (i.e., the start time plus the processing time) of job i. Then the objective is to minimize the maximum absolute difference $\max_i |C_i - d_i|$ of the completion time from the due date. This problem is formally described as follows.

Problem Scheduling-1

input: A set of n jobs $i = 1, 2, \ldots, n$ and m machines $k = 1, 2, \ldots, m$. Each job i has a due date d_i and a processing time t_{ik} if it is processed on machine k.

output: An assignment of jobs to machines and the start time of each job that minimizes $\max_i |C_i - d_i|$, where C_i is the completion time of job i.

Scheduling-1 is polynomially reducible to RPGSC by the following reduction. In the reduced RPGSC instance, each rectangle has m modes, where mode k corresponds to machine k. For each $i = 1, 2, \ldots, n$ and $k = 1, 2, \ldots, m$, we set

$$w_i^{(k)} = t_{ik}, \qquad h_i^{(k)} = 1,$$

$$p_i^{(k)}(x) = \begin{cases} -x + d_i - w_i^{(k)} & (x \le d_i - w_i^{(k)}) \\ x - d_i + w_i^{(k)} & (x > d_i - w_i^{(k)}), \end{cases} \qquad q_i^{(k)}(y) = \begin{cases} +\infty & (y < k) \\ 0 & (y = k) \\ +\infty & (y > k). \end{cases}$$

Here, $p_i^{(k)}(x)$ denotes the absolute difference of the completion time from the due date for job i and $q_i^{(k)}(y)$ means that y coordinate of job i must be k. The objective of the reduced RPGSC instance is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$. In the solution of the reduced RPGSC instance, $\mu_i(\pi)$ represents the assignment of job i and $x_i(\pi)$ represents the start time of job i.

Scheduling-2 is a scheduling problem that arises in a factory producing large building blocks. The blocks produced in the factory are very large, and, once the building block is placed in the factory, it cannot be moved until all processes on the building block 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, building blocks can be regarded as one-dimensional objects. Blocks must be placed without overlap. A schedule is determined by the place and the start time S_i of each block i. Let C_i be the completion time of the process for block i; i.e., $C_i = S_i + t_i$. Then the objective is to minimize the maximum absolute difference $\max_i\{0, s_i - S_i, C_i - d_i\}$. The problem is summarized as follows.

Problem Scheduling-2

input: A length H of the whole work space and n building blocks $I = \{1, 2, \ldots, n\}$, where each building block $i \in I$ has its length l_i, processing time t_i, ready time s_i and due date d_i.

output: The place and the start time S_i for each block i so that the maximum absolute difference $\max_i\{0, s_i - S_i, C_i - d_i\}$ is minimized, where C_i is the completion time of the processes on block i.

This problem is also polynomially reducible to RPGSC by the following reduction. In the resulting RPGSC instance, each rectangle has only one mode. For each $i = 1, 2, \ldots, n$, we set

$$w_i^{(1)} = t_i, \qquad h_i^{(1)} = l_i,$$

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

Here, $p_i^{(k)}(x)$ denotes the absolute difference from the given time window for block i, and the objective of the reduced RPGSC instance is to minimize $p_{\max}(\pi) + q_{\max}(\pi)$. In the solution of reduced RPGSC instance, $x_i(\pi)$ represents the start time S_i of processing block i and $y_i(\pi)$ represents its place in the work space.

Chapter 3

Local Search Algorithms for RPGSC

3.1 Introduction

In this chapter[1], we consider the rectangle packing problem with general spatial costs (RPGSC), which is proposed in Section 2.4, and develop metaheuristic algorithms based on local search.

As we mentioned in Chapter 2, 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. To overcome this, numerous coding schemes to represent solutions have been proposed [8, 21, 77, 90, 92].

In this chapter, we adopt the sequence pair representation [90] 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 [105, 106] 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 the area minimization problem. We

[1]The results of this chapter appear in: S. Imahori, M. Yagiura, T. Ibaraki, "Local search algorithms for the rectangle packing problem with general spatial costs," *Mathematical Programming* 97 (2003) 543–569, [63].

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 these algorithms are described in Section 3.3

We then consider various neighborhoods in Section 3.4, 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 3.5. 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.

3.2 Sequence pair

As noted before, numerous coding schemes to represent solutions have been proposed [77, 90, 92]. 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. The sequence pair [90] 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 properties of the sequence pair representation.

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 (2.4.4) to (2.4.7). 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 binary relations $\preceq_\sigma^x$ and $\preceq_\sigma^y$ by

$$\sigma_+^{-1}(i) < \sigma_+^{-1}(j) \text{ and } \sigma_-^{-1}(i) < \sigma_-^{-1}(j) \Longleftrightarrow i \preceq_\sigma^x j, \tag{3.2.1}$$

$$\sigma_+^{-1}(i) > \sigma_+^{-1}(j) \text{ and } \sigma_-^{-1}(i) < \sigma_-^{-1}(j) \Longleftrightarrow i \preceq_\sigma^y j, \tag{3.2.2}$$

for any pair i and j of rectangles. Here, the following property holds.

Property 1 [90]**:** 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 $i \neq j$.

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 $j \in I$:

$$\mu_i(\pi) = \mu_i, \tag{3.2.3}$$

$$i \preceq_\sigma^x j \Longrightarrow x_i(\pi) + w_i^{(\mu_i)} \leq x_j(\pi), \tag{3.2.4}$$

$$i \preceq_\sigma^y j \Longrightarrow y_i(\pi) + h_i^{(\mu_i)} \leq y_j(\pi). \tag{3.2.5}$$

This means that any packing $\pi \in \Pi_{\sigma,\mu}$ is feasible (i.e., no two rectangles overlap each other) 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 [90] 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 Subsection 3.3.3 (one of which runs in $O(n \log n)$ time) also give a proof of this fact.

3.3 Decoding and encoding algorithms

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

$$\begin{array}{lll} \text{RPGSC } (\sigma,\mu)\text{:} & \text{minimize} & g(p_{\max}(\pi), q_{\max}(\pi)) \\ & \text{subject to} & \pi \in \Pi_{\sigma,\mu}, \end{array} \tag{3.3.6}$$

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 and subscript representing the mode in Section 3.3 (e.g., we use w_i and Π_σ instead of $w_i^{(\mu_i(\pi))}$ and $\Pi_{\sigma,\mu}$, respectively), since the mode $\mu_i(\pi)$ of each rectangle i is fixed when we consider the decoding and encoding algorithms.

3.3.1 A decoding algorithm

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 $p_{\max}(\pi)$.

Let us define J_i^{f} (f stands for forward) 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 $\max_{j \in J_i^{\mathrm{f}} \cup \{i\}} p_j(x_j(\pi))$ subject to $x_j(\pi) + w_j \leq x_{j'}(\pi)$ for all $j, j' \in J_i^{\mathrm{f}} \cup \{i\}$ with $j \neq j'$ 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 (2.4.2) 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

$$f_i(x) = \begin{cases} \min_{x_i \leq x - w_i} p_i(x_i), & \text{if } J_i^{\mathrm{f}} = \emptyset, \\ \min_{x_i \leq x - w_i} \max\{p_i(x_i), \max_{j \in J_i^{\mathrm{f}}} f_j(x_i)\}, & \text{otherwise.} \end{cases} \tag{3.3.7}$$

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

$$x_i(\pi) = \begin{cases} \max\{x_i \mid p_i(x_i) = \min_{x_i'}\{p_i(x_i') \mid f_i(x_i' + w_i) = \min_x f_i(x)\}\}, \\ \qquad\qquad \text{if } J_i^{\mathrm{b}} = \emptyset, \\ \max\{x_i \mid p_i(x_i) = \min_{x_i'}\{p_i(x_i') \mid f_i(x_i' + w_i) = \min_x\{f_i(x) \mid x \leq r_i\}\}\}, \\ \qquad\qquad \text{otherwise,} \end{cases} \tag{3.3.8}$$

where $J_i^{\mathrm{b}} = \{j \in I \mid i \preceq_\sigma^x j\}$ (b stands for backward) and $r_i = \min_{j \in J_i^{\mathrm{b}}} x_j(\pi)$. 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 two procedures are basic operations to compute $f_i(x)$, and can be obtained 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. Hence, we utilize procedure Take-Min-Max instead of Take-Max.

If we compute $f_i(x)$ by using (3.3.7) 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 the function $\max_{j \in J_i^{\mathrm{f}}} f_j(x_i)$ in (3.3.7). 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:

$$f_l^k(x) = \begin{cases} f_{\sigma_-(l)}(x), & \text{if } \sigma_+^{-1}(\sigma_-(l)) \leq k, \\ -\infty, & \text{if } \sigma_+^{-1}(\sigma_-(l)) \geq k+1. \end{cases} \tag{3.3.9}$$

Then, we can compute $f_i(x)$ of (3.3.7) from $p_i(x)$ and $f_l^k(x)$ since

$$\max_{j \in J_i^{\mathrm{f}}} f_j(x) = \max\{f_l^k(x) \,|\, l < \sigma_-^{-1}(i) \text{ and } k = \sigma_+^{-1}(i) - 1\}. \tag{3.3.10}$$

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 $\hat{f}_l^k(x)$ for each internal node l and $k \in \{0, 1, \ldots, n\}$ as follows:

$$\hat{f}_l^k(x) = \max_{j \in T_l} f_j^k(x). \tag{3.3.11}$$

We compute $f_l^k(x)$ and $\hat{f}_l^k(x)$ from $k = 1$ to n step by step. Initially, $f_l^0(x) = -\infty$ and $\hat{f}_l^0(x) = -\infty$ 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 $F^k(x)$. Let us initialize $F^k(x) := -\infty$ 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 $F^k(x) :=$ Take-Min-Max $(\hat{f}_{l'}^{k-1}, F^k)$ (if l' is an internal node) or $F^k(x) :=$ Take-Min-Max $(f_{l'}^{k-1}, F^k)$ (if l' is a leaf) for the left child l' of l. Finally, $F^k(x)$ becomes $\max\{f_l^{k-1}(x) \,|\, l < \sigma_-^{-1}(\sigma_+(k))\}$ (see Figure 3.1 as an example). This is equal to the right hand side of (3.3.10), and then we can compute $f_i(x)$ $(= f_{l_k}^k(x))$ by (3.3.7). Next, we will compute $\hat{f}_l^k(x)$ for all internal nodes l. In this step, $\hat{f}_l^k(x) = \hat{f}_l^{k-1}(x)$ holds for each l such that $l_k \notin T_l$, and hence we should compute

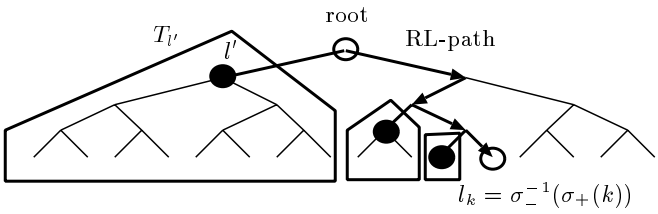


Figure 3.1: An example to compute $F^k(x)$

only $\hat{f}_l^k(x)$ such that $l_k \in T_l$, and this condition means that node l is in RL-path. Then, we compute $\hat{f}_l^k(x) :=$ Take-Min-Max $(\hat{f}_l^{k-1}, f_{l_k}^k)$ for all internal nodes l in RL-path.

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

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, $\hat{f}_l^0(x) := -\infty$ for all internal nodes and $k := 1$.

Step 3: (Compute $F^k(x)$ along RL-path.)

3.1: Let $F^k(x) := -\infty$ 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, let $l := l^*$ and return to 3.2; otherwise let $F^k(x) :=$ Take-Min-Max $(\hat{f}_{l'}^{k-1}, F^k)$ (l': internal node) or Take-Min-Max $(f_{l'}^{k-1}, F^k)$ (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 $(p_{\sigma_+(k)}, F^k)$.

Step 5: Let $\hat{f}_l^k(x) :=$ Take-Min-Max $(\hat{f}_l^{k-1}, f_{l_k}^k)$ for all internal nodes l in RL-path.

Step 6: If $k = n$, 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 2.4), 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 $\hat{f}_l^k(x)$ since $f_i(x)$ can be computed from $f_l^k(x)$ and $\hat{f}_l^k(x)$. (More details of τ will be discussed in Subsection 3.3.2.) Thus the time complexity of algorithm CMPF is $O(\tau n \log n + \delta)$.

3.3.2 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 $\hat{f}_l^k(x)$. 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 $x \geq d_i$. In this case, τ becomes $O(1)$ and 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 [58, 90, 92, 105, 106, 116], 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 [105, 106].

(2) Cost functions $p_i(x)$ are convex for all i. Then functions $f_l^k(x)$ and $\hat{f}_l^k(x)$ become convex, 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 $\hat{f}_l^k(x)$ 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 $\hat{f}_l^k(x)$ 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 $\hat{f}_l^k(x)$ 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 $\hat{f}_l^k(x)$ 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 $\hat{f}_l^k(x)$ is given by the space complexity of the upper envelope of $O(\delta)$ line segments, which is known to be $O(\delta\, \alpha(\delta, \delta))$ [4]. 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 of the three cases. Algorithm CMPF is a polynomial time algorithm under the assumption that the information of each linear piece of function $p_i(x)$ is given explicitly. However, in general, δ can be exponentially large if functions $p_i(x)$ are given implicitly, and in such cases, the algorithm proposed by Ahuja, Hochbaum and Orlin [5, 6] is more efficient for Case 2 (i.e., cost functions $p_i(x)$ are convex for all i). Note that they consider a slightly different problem and a careful transformation is necessary.

3.3.3 Transformation from a packing to a sequence pair

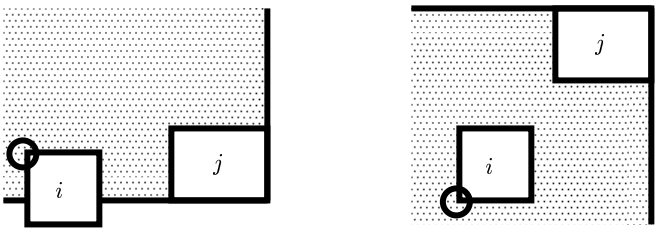
We propose an $O(n\log n)$ time algorithm to find a sequence pair $\sigma = (\sigma_+, \sigma_-)$ that satisfies $\pi \in \Pi_\sigma$ for a given packing (i.e., an encoding algorithm). This problem is independent of the spatial cost functions.

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

$$(x_i(\pi) < x_j(\pi) + w_j \text{ and } y_i(\pi) + h_i > y_j(\pi)) \Longleftrightarrow i \prec_+ j, \tag{3.3.12}$$

$$(x_i(\pi) < x_j(\pi) + w_j \text{ and } y_i(\pi) < y_j(\pi) + h_j) \Longleftrightarrow i \prec_- j. \tag{3.3.13}$$

Figure 3.2 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 $\sigma = (\sigma_+, \sigma_-)$ be a sequence pair that



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

Figure 3.2: Relationships between $\prec_+$, $\prec_-$ and coordinates of rectangles

satisfies the following two conditions:

$$i \prec_+ j \Longrightarrow \sigma_+^{-1}(i) < \sigma_+^{-1}(j), \tag{3.3.14}$$

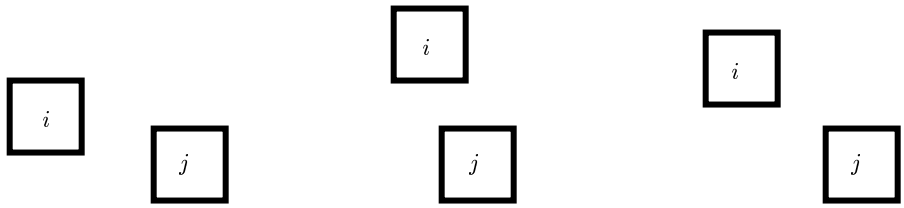
$$i \prec_- j \Longrightarrow \sigma_-^{-1}(i) < \sigma_-^{-1}(j). \tag{3.3.15}$$

Then we have the following lemma.

Lemma 1: A packing π satisfies $\pi \in \Pi_\sigma$ for any sequence pair $\sigma = (\sigma_+, \sigma_-)$ satisfying conditions (3.3.14) and (3.3.15).

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 (2.4.4)–(2.4.7) holds for i and j. For example, if $i \prec_+ j$ and $i \prec_- j$ hold, then the locations of i and j satisfy $x_i(\pi) + w_i \leq x_j(\pi)$ (see Figure 3.3 (a)). In this case, $\sigma = (\sigma_+, \sigma_-)$ satisfies $\sigma_+^{-1}(i) < \sigma_+^{-1}(j)$ and $\sigma_-^{-1}(i) < \sigma_-^{-1}(j)$ by (3.3.14) and (3.3.15), and π satisfies the implied condition (3.2.4) (i.e., $x_i(\pi) + w_i \leq x_j(\pi)$). The case with $j \prec_+ i$ and $i \prec_- j$ is similar (see Figure 3.3 (b)), and π satisfies the implied condition (3.2.5) (i.e., $y_i(\pi) + h_i \leq y_j(\pi)$). If rectangles i and j are comparable in exactly one of relations $\prec_+$ and $\prec_-$, two of the four conditions (2.4.4)–(2.4.7) 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 Figure 3.3 (c)). In this case, $\sigma = (\sigma_+, \sigma_-)$ satisfies $\sigma_+^{-1}(i) < \sigma_+^{-1}(j)$ by (3.3.14) but no restriction is imposed between $\sigma_-^{-1}(i)$ and $\sigma_-^{-1}(j)$. However, since π satisfies both the implied conditions (3.2.4) and (3.2.5) (corresponding to $\sigma_-^{-1}(i) < \sigma_-^{-1}(j)$ and $\sigma_-^{-1}(i) > \sigma_-^{-1}(j)$, respectively), π does not contradict with conditions (3.2.4) and (3.2.5). These arguments prove that of $\pi \in \Pi_\sigma$ holds. ■

We first explain a simple $O(n^2)$ time algorithm to compute a sequence pair $\sigma = (\sigma_+, \sigma_-)$



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

Figure 3.3: Relationships between packing π and relations $\prec_+$, $\prec_-$

satisfying (3.3.14) and (3.3.15) 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 $j \in I_i^{\mathrm{LA}} \Longleftrightarrow i \in I_j^{\mathrm{RB}}$ 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 permutation 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. First, we define the set $I_i^{\mathrm{L}} \subseteq I$ (L stands for left) for each $i \in I$ as follows:

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

Note that we permit that the line segment $((x_j(\pi) + w_j, y), (x_i(\pi), y))$ has length 0. Figure 3.4 illustrates set I_i^L and locations of rectangles ($j_2, j_4 \in I_i^L$ and $j_1, j_3 \notin I_i^L$). We also

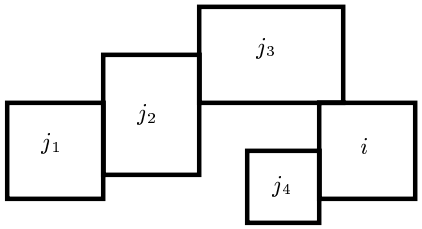


Figure 3.4: A relationship between set I_i^L and locations of rectangles

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 [9].

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 pair of i and 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 these operations.

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 [9]. 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 permutation 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 $S_{\mathrm{L,A}}^0$ and delete it from S and $S_{\mathrm{L,A}}^0$. 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 $O(n \log n)$.

3.3.4 Another decoding algorithm

In this section, we describe algorithm Compute-Better-Packing (CBP), which computes a packing π from a given sequence pair σ. Although the packing π computed by CBP may not satisfy $\pi \in \Pi_\sigma$, it is not worse than any packing in Π_σ in terms of the objective value. Note that, Nagao et al. [91] proposed a decoding algorithm which has similar properties to ours, i.e., computes a better or equivalent packing π than any packing which satisfies all the given constraints on relative locations of a sequence pair σ. Their algorithm, however, is specialized to the area minimization problem, and our algorithm utilizes different strategies to it. While algorithm CMPF in Subsection 3.3.1 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

$$x_{\sigma_-(1)}(\pi^0) = 0, \tag{3.3.16}$$

$$x_{\sigma_-(l)}(\pi^0) = x_{\sigma_-(l-1)}(\pi^0) + w_{\sigma_-(l-1)},\ l = 2, 3, \ldots, n. \tag{3.3.17}$$

The obtained packing π^0 is in Π_σ and $q_{\max}(\pi^0) \leq q_{\max}(\pi)$ holds for all $\pi \in \Pi_\sigma$. Then, we compute a packing π that minimizes $p_{\max}(\pi)$ among those satisfying

$$j \in I^{\mathrm{L}}_i \Longrightarrow x_j(\pi) + w_j \leq x_i(\pi) \text{ for all } i \in I, \tag{3.3.18}$$

where I^{L}_i (see the previous subsection for the definition) is defined on π^0 and $y_i(\pi) = y_i(\pi^0)$ holds for all $i \in I$. Since $j \in I^{\mathrm{L}}_i \Longrightarrow j \preceq^x_\sigma i$ holds for all i and j, constraints (3.3.18) are not stronger than (3.2.4). That is, the feasible region of the problem considered here contains that of RPGSC(σ, μ). Hence, $p_{\max}(\pi)$ obtained by CBP is not worse than that obtained by CMPF. Let us define $\tilde{f}_i(x)$ as follows:

$$\tilde{f}_i(x) = \begin{cases} \min_{x_i \leq x - w_i} p_i(x_i), & \text{if } I^{\mathrm{L}}_i = \emptyset, \\ \min_{x_i \leq x - w_i} \max\{p_i(x_i), \max_{j \in I^{\mathrm{L}}_i} \tilde{f}_j(x_i)\}, & \text{otherwise,} \end{cases} \tag{3.3.19}$$

and the minimum penalty value $p_{\max}(\pi)$ can be obtained by

$$p_{\max}(\pi) = \max_{i \in I} \min_{x} \tilde{f}_i(x). \tag{3.3.20}$$

The horizontal coordinate x_i of each rectangle i can be computed by the similar computation to (3.3.8) as explained in Subsection 3.3.1. As we mentioned in the previous subsection, $\sum_i |I_i^{\mathrm{L}}| = O(n)$ holds, and hence procedure Take-Min-Max is called $O(n)$ times in the recursion of (3.3.19).

The packing π computed by algorithm CBP may not satisfy $\pi \in \Pi_\sigma$, where σ is the sequence pair to which CBP is applied. In this case, we can find a sequence pair σ' which satisfies $\pi \in \Pi_{\sigma'}$ in $O(n \log n)$ time by applying the encoding algorithm P2SP-2 proposed in the previous section 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 sequence pair σ' satisfying $\pi \in \Pi_{\sigma'}$.

The time complexity of algorithm CBP (even if the encoding algorithm is included) 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.

3.4 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 Subsection 3.4.1. After giving the definition of critical paths in Subsection 3.4.2, we explain four types of neighborhoods based on critical paths in Subsection 3.4.3. In Subsection 3.4.4, we explain frameworks of the proposed metaheuristic algorithms.

3.4.1 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. The search space of our algorithm 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 Subsection 3.3.1 or algorithm CBP in Subsection 3.3.4. 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).

3.4.2 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 $S, T, P \subseteq I$ as follows:

$$\begin{aligned} &V = I, \\ &(i,j) \in E \Longleftrightarrow x_i(\pi) + w_i^{(\mu_i(\pi))} = x_j(\pi) \text{ and } i \preceq_\sigma^x j, \\ &S = \{i \in I \mid p_i(x_i(\pi) - \varepsilon) \geq p_{\max}(\pi) \text{ for an arbitrarily small } \varepsilon > 0\}, \\ &T = \{i \in I \mid p_i(x_i(\pi) + \varepsilon) \geq p_{\max}(\pi) \text{ for an arbitrarily small } \varepsilon > 0\}, \\ &P = \{i \in I \mid p_i(x_i(\pi)) = p_{\max}(\pi)\}. \end{aligned} \tag{3.4.21}$$

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 at least one vertex $v \in P$ exists in this path (including the end vertices). For any packing π obtained by our decoding algorithms, S, T and P are nonempty and there is at least one critical path for each direction. Critical paths have an important property: $p_{\max}(\pi)$ cannot be decreased without breaking all critical paths of

the x direction. Therefore we introduce neighborhoods, which are based on critical paths, in the next subsection.

Now, we consider an algorithm to compute critical paths. It is easy to find all pairs of rectangles which are adjacent on critical paths in $O(n^2 + \delta)$ time with a simple algorithm. Here, we propose an algorithm to find critical paths for horizontal direction which runs faster than the simple one. An outline of our algorithm is described as follows.

Algorithm: Find-Critical-Paths-x

Step 1: Compute the sets S, T and P.

Step 2: Construct the directed graph $G = (V, E)$ of (3.4.21).

Step 3: Find critical paths on $G = (V, E)$.

We consider each step and analyze its time complexity. In Step 1, we compute $p_i^{(\mu_i(\pi))}(x_i)$ and the gradient of $p_i^{(\mu_i(\pi))}(x)$ at x_i for all $i \in I$. If x_i is an end point of a linear segment of function $p_i^{(\mu_i(\pi))}(x)$, we check both segments which have end point x_i. It takes $O(\delta_i)$ time to compute this for $i \in I$. Hence, the total computation time for Step 1 is $O(\delta)$. Next, we consider Step 2. It is easy to construct $G = (V, E)$ in $O(n^2)$ time, since condition (3.4.21) can be checked in $O(1)$ time for each pair i and j of n rectangles. A directed graph has $O(n^2)$ edges in the worst case; however, there are much less edges in G in general. We propose algorithm Construct-Directed-Graph (CDG) to construct $G = (V, E)$ much faster than $O(n^2)$ time in practice.

This algorithm is based on hashing. We define a set Q_x as the rectangles i such that $x_i(\pi) + w_i^{(\mu_i(\pi))} = x$. Then, for each rectangle i, we check whether $Q_{x_i(\pi)} = \emptyset$ or not by using the hash table. If $Q_{x_i(\pi)} \neq \emptyset$ holds, we check for each $j \in Q_{x_i(\pi)}$ whether $j \preceq_\sigma^x i$ holds or not and output necessary edges accordingly. We use the balanced binary search tree to keep the set Q_x to enumerate those j satisfying $j \preceq_\sigma^x i$ efficiently, where the key of the binary search tree is $\sigma_-^{-1}(j)$. The algorithm is formally described as follows.

Algorithm: Construct-Directed-Graph (CDG)

Step 1: Set $l := 1$ and prepare an empty hash table (i.e., set $Q_x := \emptyset$ for all x).

Step 2: Set $i := \sigma_+(l)$.

Step 3: Execute "member $x_i(\pi)$" to the hash table to check if $Q_{x_i(\pi)}$ is empty or not. If $Q_{x_i(\pi)} \neq \emptyset$ holds, go to Step 4; otherwise go to Step 5.

Step 4: Output directed edges (j, i) for each $j \in Q_{x_i(\pi)}$ such that $\sigma_-^{-1}(j) < \sigma_-^{-1}(i)$ holds.

Step 5: Insert $x_i(\pi)+w_i^{(\mu_i(\pi))}$ into the hash table. Then we set $Q_x := Q_x \cup \{i\}$ for $x = x_i(\pi) + w_i^{(\mu_i(\pi))}$.

Step 6: If $l = n$ holds, stop; otherwise set $l := l + 1$ and return to Step 2.

We analyze the running time of algorithm CDG. Since the number of insertions is n, we set the size of the hash table to $2n$ as suggested in [73]; hence Step 1 is possible in $O(n)$ time. For simplicity, we assume that the insertion and member operations for the hash table are possible in $O(1)$ time. Then the member and insertion operations in Steps 3 and 5 are possible in $O(n)$ time in total. Step 4 (for a fixed i) is executed by first inserting i into the binary search tree, then output all $j \in Q_{x_i(\pi)}$ satisfying $\sigma_-^{-1}(j) < \sigma_-^{-1}(i)$ by using the inorder traverse of the tree, and finally deleting i from the tree. This takes $O(\log \eta + |E_i|)$ time, where η is the maximum size of Q_x (hence $\eta \leq n$ holds) and E_i is the set of edges (j, i) output in Step 4 for a fixed i. Therefore, algorithm CDG runs in $O(n \log \eta + |E|)$ time in total.

Now, we consider Step 3 of algorithm Find-Critical-Paths-x. We can find all pairs of rectangles adjacent on critical paths by tracing the resulting directed graph. This can be executed in $O(|V| + |E|)$ time.

In summary, the total time requirement of algorithm Find-Critical-Paths-x is $O(\delta + n \log \eta + |E|)$. In many cases, $\delta = O(n)$, $\eta = O(1)$ and $|E| = O(n)$ hold, and hence the time complexity becomes $O(n)$ in practice.

3.4.3 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.

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_-, the constraints related to the

binary relations $\preceq_\sigma^x$ 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 Subsection 3.5.4. We call the swap neighborhood without any reductions as 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). We therefore choose at least one rectangle in a swap from those rectangles on 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 on 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 on the critical path of x direction and j is not, and $w_i^{(\mu_i(\pi))} > w_j^{(\mu_j(\pi))}$ holds.
- i is on the critical path of y direction and j is not, and $h_i^{(\mu_i(\pi))} > h_j^{(\mu_j(\pi))}$ holds.

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.

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 $(\neq i)$ arbitrarily 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

rectangle 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 Subsection 3.5.4

(1) We restrict the rectangles i 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 on critical paths. We call this ShiftCri.

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 on 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 Figure 3.5 for illustration). This

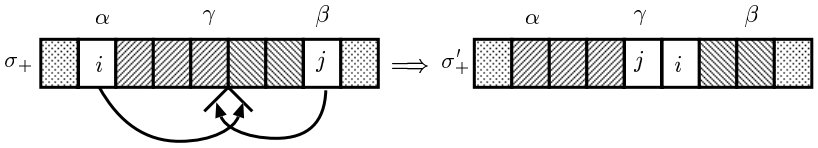
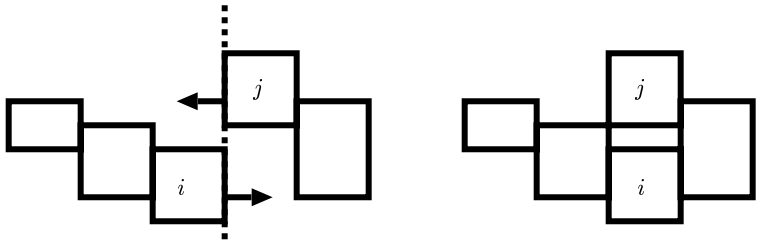


Figure 3.5: An example of changing σ_+ to σ'_+ with a swap* operation

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 on 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 Figure 3.5):

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

Then, we can change from a packing in Figure 3.6 (a) to a packing in Figure 3.6 (b) with a swap* operation on i and j. We consider all possible i and j, and all γ satisfying



(a) $i \preceq_\sigma^x j$ (b) $i \preceq_\sigma^y j$

Figure 3.6: An example showing the effect of a swap* operation

$\alpha \leq \gamma < \beta$; 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*.

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.

Combination of some neighborhoods

It is often effective to combine more than one neighborhood. In the computational experiments of Section 3.5, 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.

3.4.4 Metaheuristics

We explain three metaheuristic algorithms, all of which are based on local search. These will be used in our computational experiments in Section 3.5. Computation of these algorithms continues until the prespecified computational time is reached.

(1) The *random multi-start local search* (MLS) [70]: 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) [68]: ILS is a variant of MLS, in which initial solutions are generated by slightly perturbing a good solution $(\sigma_{\rm seed}, \mu_{\rm seed})$ found during the previous search. In our ILS, $(\sigma_{\rm seed}, \mu_{\rm seed})$ is defined to be the best solution obtained so far with respect to function *eval*. The next initial solution is then generated from $(\sigma_{\rm seed}, \mu_{\rm seed})$ by applying swap, shift or change mode operations a few times randomly.

(3) The algorithm WALK is similar to algorithm WALK-SAT [103] 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 on 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).

3.5 Computational experiments

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 Subsection 3.5.1. In Subsections 3.5.2 and 3.5.3, we examine the performance of our decoding algorithms proposed in Subsections 3.3.1 and 3.3.4, and encoding algorithms in Subsection 3.3.3. We then report computational results of local search with various implementations using different neighborhoods in Subsection 3.5.4. Computational results of various metaheuristic algorithms are reported in Subsection 3.5.5. In Subsection 3.5.6, we compare our algorithms with other existing heuristic algorithms for both the rectangle packing problem and the scheduling problem.

3.5.1 Test problems and their instances

We explain two problems and their instances, which are used in our computational experiments. The definitions of these problems have already been given in Subsection 2.4.2, however, we explain them again for readability. All instances can be obtained electronically from `http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing/`.

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 rectangular bin (large object) that contains all given rectangles. This

problem was considered in many papers including [90, 92, 105, 106]. 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

$$w_i^{(1)} = w_i, \qquad h_i^{(1)} = h_i,$$
$$w_i^{(2)} = h_i, \qquad h_i^{(2)} = w_i.$$

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

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

The objective of the resulting RPGSC instance is to minimize $p_{\max}(\pi) \cdot q_{\max}(\pi)$ (i.e., the area of the large object 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 [90, 92]. 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.

The scheduling problem of large building blocks

This is a scheduling problem that arises in a factory producing large building blocks. The blocks produced in the factory are very large, and once the building block is placed in the factory, it cannot be moved until all processes on the building block 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, building blocks can be regarded as the one-dimensional objects. Blocks must be placed without overlap. A schedule is determined by the place and the start time S_i of each block i. Let C_i be the completion time of the process for block i; i.e., $C_i = S_i + t_i$. Then the objective is to minimize the maximum absolute difference $\max\{0, s_i - S_i, C_i - d_i\}$.

This problem can also be formulated as RPGSC, in which each rectangle has only one

mode. For $i = 1, 2, \ldots, n$, let

$$w_i^{(1)} = t_i, \qquad h_i^{(1)} = l_i,$$

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

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 S_i 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 ready time, due date and work space for all building blocks).

3.5.2 Decoding algorithms

In this section, we examine the performance of our decoding algorithms proposed in Subsections 3.3.1 and 3.3.4. We compare three algorithms: (1) an $O(\tau n^2 + \delta)$ time naive algorithm in Subsection 3.3.1 (denoted NAIVE), (2) algorithm CMPF in Subsection 3.3.1 whose time complexity is $O(\tau n \log n + \delta)$ and (3) algorithm CBP in Subsection 3.3.4 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 3.1. 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 Subsection 3.3.2-(1) (i.e., τ is $O(1)$), and "general" means that rectangles have general cost functions, each with a few segments.

From Table 3.1, 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. The reason is as follows: CBP can have a larger coefficient on the $n \log n$ term than CMPF, if τ is a small constant, since CBP uses the plane sweep

Table 3.1: 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}

Table 3.2: 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

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 of the area minimization problem: ami49, rp100, pcb146, rp200 and pcb500. Algorithms CMPF and CBP are incorporated in the local search algorithms, and results are shown in Table 3.2. Column "value" shows the average of the following ratio,

$$100 \times \frac{\text{(the lower bound of the objective function)}}{\text{(the objective value of the local optimum)}},$$

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

3.3, 3.4 and 3.5. Note that we use neighborhood Union (defined in Subsection 3.4.3) in each LS. From Table 3.2, 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 Subsections 3.5.4, 3.5.5 and 3.5.6.

3.5.3 Encoding algorithms

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) proposed in Subsection 3.3.3 by applying them to various instances in Subsection 3.5.1. 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 Subsections 3.5.4, 3.5.5 and 3.5.6.

3.5.4 Neighborhoods

The following eight types of neighborhoods discussed in Subsection 3.4.3 were computationally compared from the view point of the performance of the resulting local search algorithms LS, on test instances of the area minimization problem in Subsection 3.5.1.

- SwapAll (the swap neighborhood).
- SwapMax (a reduced swap neighborhood).
- SwapCri (the other reduced swap neighborhood).
- ShiftAll (the shift neighborhood).
- ShiftMax (a reduced shift neighborhood).
- ShiftCri (the other reduced shift neighborhood).
- Swap* (the swap* neighborhood).
- Union (the combination of SwapCri, ShiftMax and Swap*).

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 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 3.3 for five test instances. The mark "†" indicates that LS is forced to stop by the time limit. We can observe from Table 3.3 that SwapAll is the best neighborhood with respect to the quality of solutions, but takes

Table 3.3: 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†

Table 3.4: 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

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, those 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 Subsections 3.5.5 and 3.5.6.

3.5.5 Metaheuristics

We compared the three metaheuristic algorithms MLS, ILS and WALK described in Subsection 3.4.4 on five test instances of the area minimization problem. We ran each algorithm until 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 3.4. We can observe that ILS found better solutions than other two algorithms for many instances.

3.5.6 Comparison with other algorithms

Finally, we compared the performance of our algorithm ILS-CBP with other existing heuristic algorithms, on various instances of the rectangle packing problem and the scheduling problem explained in Subsection 3.5.1. First, we compared our algorithm ILS-CBP with two heuristic algorithms for the area minimization problem: (1) A simulated annealing algorithm with the BSG (bounded sliceline grid) coding scheme by Nakatake, Fujiyoshi, Murata and Kajitani (denoted SA-BSG) [92] and (2) a simulated annealing algorithm with

Table 3.5: Comparison with other methods for the rectangle packing problem

	SA-BSG		SA-SP		ILS-CBP	
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

the sequence pair coding scheme by Murata, Fujiyoshi, Nakatake and Kajitani (denoted SA-SP) [90]. We used five test instances of the area minimization problem, ami49, rp100, pcb146, rp200 and pcb500. Note that both algorithms SA-BSG and SA-SP 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 3.5. From the table, we can observe that our 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 numerous 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 Subsection 3.5.1 with a tabu search algorithm developed for the resource constrained project scheduling problem by Nonobe and Ibaraki (denoted TS-RCPSP) [93]. To solve the problem by TS-RCPSP, we reformulated the problem as a project scheduling problem in the way as described in [56]. Note that TS-RCPSP 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 five test instances described in Subsection 3.5.1 and results are shown in Table 3.6. Column "ratio" shows (# of successful trials)/(# of trials). Column "time" shows the average computational time (in seconds) to find an optimal solution in successful trials. From Table 3.6, we can observe that ILS-CBP is

Table 3.6: Comparison with algorithm TS-RCPSP on the scheduling problem

	TS-RCPSP		ILS-CBP	
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

superior to TS-RCPSP in both of the solution quality and the computational time for all instances. These results also exhibit good prospects of our algorithm.

3.6 Conclusion

In this chapter, we considered the rectangle packing problem with spatial costs, which is general in that it contains various types of cutting and 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 functions and the spatial cost functions, respectively. This algorithm generalizes the results of [105, 106] 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 [105, 106], if applied to the case of the area minimization problem.

These algorithms were then incorporated in the local search and metaheuristic algorithms. We defined critical paths for the x and y directions of 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.

Chapter 4

Improved Local Search Algorithms for RPGSC

4.1 Introduction

In this chapter[1], we continue to consider the rectangle packing problem with general spatial costs (RPGSC) which was proposed in Section 2.4. 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.

As we mentioned in the previous chapters, if we search directly the x and y coordinates and the mode of each rectangle, an effective search will be difficult since the number of solutions is uncountably many and eliminating overlap of rectangles is not easy. We therefore adopt the sequence pair [90] as the coding scheme in our local search algorithm for RPGSC. A solution is coded as a pair of permutations of n rectangles and a vector specifying the modes of all rectangles. In Chapter 3, we proposed decoding algorithms, CMPF and CBP, based on dynamic programming to obtain an optimal packing under the constraint specified by the coded solution. Algorithm CMPF was a generalization of the algorithms proposed in [105, 106] so that general spatial costs can be handled. The running time of this algorithm was $O(n \log n)$ if applied to the cases of the area minimization, strip packing, two-dimensional knapsack and so on.

In this chapter, we propose new decoding algorithms to evaluate all coded solutions in various neighborhoods. The amortized computational time of these algorithms per

[1]The results of this chapter appear in: S. Imahori, M. Yagiura, T. Ibaraki, "Improved local search algorithms for the rectangle packing problem with general spatial costs," submitted for publication (available at http://www-or.amp.i.kyoto-u.ac.jp/members/imahori/packing) [64].

one solution is $O(1)$ or $O(\log n)$ depending on the neighborhood, if applied to special cases including the area minimization, strip packing, two-dimensional knapsack and so on. These algorithms can not produce a packing, but only compute the objective value of the packing; however, the speed up of such evaluation is essentially important, since we must evaluate a large number of coded solutions in local search and metaheuristic algorithms.

In Section 4.2, we consider various neighborhoods which are used in the local search algorithms for finding good coded solutions. The efficiency of new decoding algorithms strongly depends on the neighborhood structure. We also make use of the critical path that corresponds to the bottleneck of the current solution (see Subsection 3.4.2 for the definition) to reduce the sizes of various neighborhoods. The local search algorithms based on these neighborhoods are then incorporated in metaheuristic algorithms such as the random multi-start local search (MLS) and the iterated local search (ILS).

The details of the proposed decoding algorithms are described in Section 4.3. In these algorithms, we utilize the common structure among all of the neighborhood solutions and evaluate those solutions based on dynamic programming.

The computational results are reported in Section 4.4. We examine the performance of proposed decoding algorithms to compare with three decoding algorithms proposed in the previous chapter. We also compare our algorithms with other existing heuristic algorithms for some variants of the rectangle packing problem and a real-world scheduling problem.

4.2 Neighborhoods for local search

In this section, we propose various types of neighborhoods and design local search and metaheuristic algorithms to find good coded solutions (σ, μ). We first explain the framework of our local search in Subsection 4.2.1, and then propose various types of neighborhoods based on the critical paths in subsequent subsections. As we mentioned in Chapters 1 and 3, if LS is applied only once, many solutions of better quality may remain unvisited in the search space. To overcome this, the local search algorithm based on various neighborhoods are then incorporated in metaheuristic algorithms such as MLS and ILS. In our ILS algorithm, we use the best locally optimal solution obtained so far (if there exist ties, we use the solution obtained most recently) as $(\sigma_{\mathrm{seed}}, \mu_{\mathrm{seed}})$, and apply random swap operations (i.e., exchange the positions of randomly chosen two rectangles in both permutations) ν times on $(\sigma_{\mathrm{seed}}, \mu_{\mathrm{seed}})$ to generate a new initial solution. Here, ν is a parameter and we set it one, two or three at random. In preliminary experiments, we observed that ILS was generally superior to MLS, and hence we will use ILS as the framework of our metaheuristic algorithm in Section 4.4.

4.2.1 Framework of local search

The frame work of our local search is basically similar to that in Subsection 3.4.1, and we omit the thorough explanation here. In this subsection, we only explain some points different from the previous one.

First, a solution (σ, μ) is evaluated by the objective value of an optimal packing $\pi \in \Pi_{\sigma,\mu}$, which can be computed efficiently by decoding algorithms in Section 4.3. However, to break ties, we also compute the number of rectangles which are on the critical paths, and put a higher priority on a packing with a smaller number of rectangles on the critical paths.

Then, we use the following three types of neighborhoods, called *shift*, *two-shifts* and *change mode* in our local search. Shift and change mode are more or less standard neighborhoods which we also utilized in the previous chapter. Two-shifts is an extension of the shift neighborhood which includes the swap neighborhood, another standard neighborhood.

4.2.2 Shift neighborhood

A shift operation changes the position of one rectangle i to another position in both σ_+ and σ_-. The *shift* neighborhood is defined to be the set of all solutions obtainable from the current solution by a shift operation. Its size is $O(n^3)$. To reduce the size of shift neighborhood, we use the information of the critical paths, i.e., we shift only those rectangles i located on the critical paths. It is easy to show that a solution is locally optimal in the original neighborhood if no improved solution is found in the reduced neighborhood. In this sense, we can reduce the size of shift neighborhood without sacrificing the solution quality. In our implementation, we further restrict the positions, where a shifted rectangle i is placed, in the following three manners.

(1) The position of the shifted rectangle is changed only in one permutation (σ_+ or σ_-). We call this a *single shift operation*.

(2) The positions, where the shifted rectangle i is inserted, are determined by the following rule: We choose one rectangle j ($\neq i$) arbitrarily and insert i before or after j in both σ_+ and σ_-, thereby four insertion positions of i are examined for each j (see Figure 4.1). 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. We call this a *limited double shift operation*.

(3) We insert rectangle i to positions close to the current positions of i in two permutations. To control the size of this neighborhood, we limit the distance from the current position to up to $\lceil a\sqrt{n} \rceil$ in each permutation, where a is a parameter. (We set $a := 1$

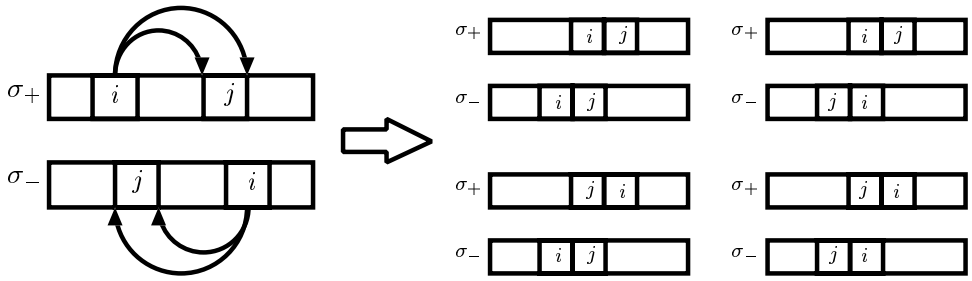


Figure 4.1: An example of the limited double shift operation

in our computational experiments.) The size of this neighborhood becomes $O(a^2 n)$ for each i. We call this a *near-place double shift operation*.

For convenience, we denote each of these operations a limited shift operation, and the set of solutions obtainable by a limited shift operation is called the *limited shift* neighborhood. The size of this neighborhood (based on three kinds of operations) is $O(cn)$, where c is the number of rectangles on the critical paths.

4.2.3 Two-shifts neighborhood

A two-shifts operation changes the positions of two rectangles i and j to another positions in σ_+ and σ_-. The size of *two-shifts* neighborhood is too large to search efficiently, and hence we propose some reduction methods for this neighborhood.

First, we restrict the rectangles i and j to be shifted. At least one rectangle must be chosen from the rectangles on critical paths to improve the current solution. However, since choosing both rectangles from critical paths appears too restrictive, we choose one rectangle from critical paths and another arbitrarily.

Next, we restrict the insertion positions of the rectangles i and j to those determined by one of the following rules:

(1) First, we exchange the positions of two rectangles i and j in both permutations. Then, we shift the position of only one rectangle i to another position in σ_+ and σ_- by a limited shift operation proposed in the previous subsection. We call this a *swap-and-shift operation*.

(2) We remove two rectangles i and j from permutations and insert them one by one in greedy fashion. First we choose a pair of permutations consisting of $n-1$ rectangles $I \setminus \{j\}$ among those obtainable by a limited shift operation of rectangle i, where $n-2$ rectangles in $I \setminus \{i, j\}$ are fixed. Then rectangle j is inserted into any position by a limited shift operation of rectangle j. We call this a *greedy two-shifts operation*.

The set of solutions obtainable by one of these operations is called the *limited two-shifts* neighborhood. The size of this neighborhood is $O(cn^2)$.

4.2.4 Change mode neighborhood

A change mode operation changes the mode of one selected rectangle. This is the only operation applied to the vector μ. We combine this operation to shift and two-shifts operations, i.e., when we insert a rectangle to permutations, we may change its mode.

4.3 Decoding algorithms

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

$$\begin{array}{lll} \text{RPGSC } (\sigma,\mu)\text{:} & \text{minimize} & g(p_{\max}(\pi), q_{\max}(\pi)) \\ & \text{subject to} & \pi \in \Pi_{\sigma,\mu}, \end{array} \tag{4.3.1}$$

and propose dynamic programming algorithms to compute the objective value of an optimal packing efficiently. We first review a basic decoding algorithm CMPF (Compute-Minimum-Penalty-Function) proposed in Subsection 3.3.1 for solving RPGSC(σ, μ). This algorithm is also used in our new local search algorithm.

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

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

$f_i(x)$: the minimum value of $\max_{j \in J_i^{\mathrm{f}} \cup \{i\}} p_j(x_j(\pi))$ subject to $x_j(\pi) + w_j \leq x_{j'}(\pi)$ for all $j, j' \in J_i^{\mathrm{f}} \cup \{i\}$ with $j \neq j'$ 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 (2.4.2) 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

$$f_i(x) = \begin{cases} \min_{x_i \leq x - w_i} p_i(x_i), & \text{if } J_i^{\mathrm{f}} = \emptyset, \\ \min_{x_i \leq x - w_i} \max\{p_i(x_i), \max_{j \in J_i^{\mathrm{f}}} f_j(x_i)\}, & \text{otherwise.} \end{cases} \tag{4.3.2}$$

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

$$x_i(\pi) = \begin{cases} \max\{x_i \,|\, p_i(x_i) = \min_{x_i'}\{p_i(x_i') \,|\, f_i(x_i' + w_i) = \min_x f_i(x)\}\}, & \\ \qquad\qquad \text{if } J_i^{\mathrm{b}} = \emptyset, & \\ \max\{x_i \,|\, p_i(x_i) = \min_{x_i'}\{p_i(x_i') \,|\, f_i(x_i' + w_i) = \min_x\{f_i(x) \,|\, x \leq r_i\}\}\}, & \\ \qquad\qquad \text{otherwise,} & \end{cases} \tag{4.3.3}$$

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

Computational time of this algorithm is $O(\tau n \log n)$ which depends on the space complexity τ (i.e., the number of linear pieces) of the minimum penalty functions $f_i(x)$. In many important special cases, such as the area minimization, strip packing, two-dimensional knapsack and so on, the spatial cost functions $p_i^{(\mu_i)}(x)$ are simple and τ becomes $O(1)$. In more general cases of $p_i^{(\mu_i)}(x)$, τ is $O(n)$ in many realistic applications. More detailed analysis of τ was discussed in Subsection 3.3.2.

We then propose new decoding algorithms for various neighborhoods in Subsections 4.3.1 and 4.3.2. We will explain algorithms to minimize $p_{\max}(\pi)$ only. Algorithms to minimize $q_{\max}(\pi)$ can be similarly defined and we can minimize the objective value of RPGSC(σ, μ) by applying these algorithms to the x and y coordinates independently.

4.3.1 Evaluating shift moves

We propose an algorithm to evaluate solutions in the shift neighborhood where the current solution is (σ, μ) and rectangle i will be shifted. We call this algorithm Evaluate-Shift-Moves (ESM), which is described as follows.

Let $\tilde{I} = I - \{i\}$ and $(\tilde{\sigma}, \tilde{\mu})$ be the coded solution obtained from the current solution (σ, μ) by removing a rectangle i, and $\tilde{\pi}$ be a packing of rectangles in $\tilde{I}$ such that $\tilde{\pi} \in \Pi_{\tilde{\sigma},\tilde{\mu}}$. We can compute an optimal packing $\tilde{\pi}^* \in \Pi_{\tilde{\sigma},\tilde{\mu}}$ from $(\tilde{\sigma}, \tilde{\mu})$ by algorithm CMPF. If a new packing π does not have the rectangle i on its critical paths of x direction, the minimum penalty value $p_{\max}(\pi)$ is equal to $p_{\max}(\tilde{\pi}^*)$.

We then compute the minimum penalty value under the influence of rectangle i. For this, let us define $\tilde{J}^{\rm f}_{\alpha,\beta}$, $\tilde{J}^{\rm b}_{\alpha,\beta}$, $\tilde{f}_{\alpha,\beta}(x)$ and $\tilde{b}_{\alpha,\beta}(x)$ for each pair α and β such that $1 \leq \alpha, \beta \leq n$.

$$\tilde{J}^{\rm f}_{\alpha,\beta} = \{j \in \tilde{I} \,|\, \tilde{\sigma}_+^{-1}(j) < \alpha, \tilde{\sigma}_-^{-1}(j) < \beta\},$$

$$\tilde{J}^{\rm b}_{\alpha,\beta} = \{j \in \tilde{I} \,|\, \tilde{\sigma}_+^{-1}(j) \geq \alpha, \tilde{\sigma}_-^{-1}(j) \geq \beta\},$$

$\tilde{f}_{\alpha,\beta}(x)$: the minimum value of $\max_{j \in \tilde{J}^{\rm f}_{\alpha,\beta}} p_j^{(\tilde{\mu}_j)}(x_j(\tilde{\pi}))$ subject to $x_j(\tilde{\pi}) + w_j^{(\tilde{\mu}_j)} \leq x, \forall j \in \tilde{J}^{\rm f}_{\alpha,\beta}$,

$\tilde{b}_{\alpha,\beta}(x)$: the minimum value of $\max_{j \in \tilde{J}^{\rm b}_{\alpha,\beta}} p_j^{(\tilde{\mu}_j)}(x_j(\tilde{\pi}))$ subject to $x_j(\tilde{\pi}) \geq x, \forall j \in \tilde{J}^{\rm b}_{\alpha,\beta}$.

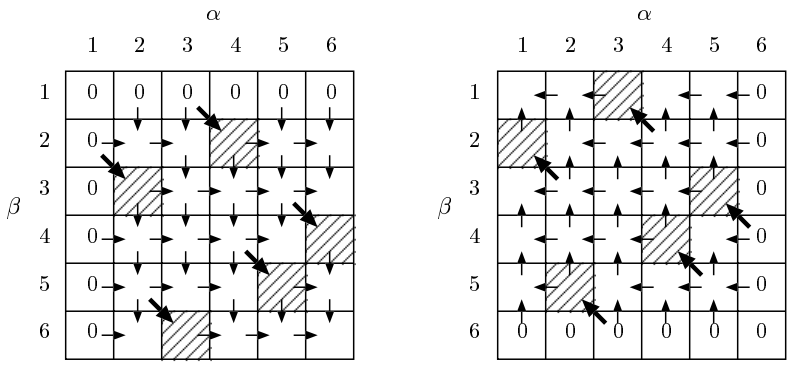
Then, by the idea of dynamic programming, $\tilde{f}_{\alpha,\beta}(x)$ (respectively, $\tilde{b}_{\alpha,\beta}(x)$) can be com-

puted by

$$\tilde{f}_{\alpha,\beta}(x) = \begin{cases} 0, & \text{if } \alpha = 1 \text{ or } \beta = 1, \\ \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha,\beta-1}(x)\}, & \text{if } \tilde{\sigma}_+(\alpha-1) \neq \tilde{\sigma}_-(\beta-1), \\ \min_{t \le x - w_j^{(\tilde{\mu}_j)}} \{\max(p_j^{(\tilde{\mu}_j)}(t), \tilde{f}_{\alpha-1,\beta-1}(t))\}, & \text{if } \tilde{\sigma}_+(\alpha-1) = \tilde{\sigma}_-(\beta-1) = j, \end{cases} \tag{4.3.4}$$

$$\tilde{b}_{\alpha,\beta}(x) = \begin{cases} 0, & \text{if } \alpha = n \text{ or } \beta = n, \\ \max\{\tilde{b}_{\alpha+1,\beta}(x), \tilde{b}_{\alpha,\beta+1}(x)\}, & \text{if } \tilde{\sigma}_+(\alpha) \neq \tilde{\sigma}_-(\beta), \\ \min_{t \ge x}\{\max(p_j^{(\tilde{\mu}_j)}(t), \tilde{b}_{\alpha+1,\beta+1}(t + w_j^{(\tilde{\mu}_j)}))\}, & \text{if } \tilde{\sigma}_+(\alpha) = \tilde{\sigma}_-(\beta) = j, \end{cases} \tag{4.3.5}$$

for all pairs of $\alpha = 1, 2, \ldots, n$ and $\beta = 1, 2, \ldots, n$ (resp., all pairs of $\alpha = n, n-1, \ldots, 1$ and $\beta = n, n-1, \ldots, 1$). The above computation is illustrated through an example in Figure 4.2. Each box in this figure corresponds to $\tilde{f}_{\alpha,\beta}(x)$ (resp., $\tilde{b}_{\alpha,\beta}(x)$) for each pair of



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

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

Figure 4.2: An example of computing $\tilde{f}_{\alpha,\beta}(x)$ and $\tilde{b}_{\alpha,\beta}(x)$

α and β, and arrows mean how to compute each function. That is, the function of each shaded box is computed by the third formula of (4.3.4) (resp., (4.3.5)). After computing these, we can obtain the minimum penalty value $p_{\max}(\pi)$ of the coded solution obtained by inserting the removed rectangle i into the αth position of $\tilde{\sigma}_+$ and the βth position of $\tilde{\sigma}_-$ with mode k, by

$$p_{\max}(\pi) = \max\{p_{\max}(\tilde{\pi}^*), \min_t \max(\tilde{f}_{\alpha,\beta}(t), p_i^{(k)}(t), \tilde{b}_{\alpha,\beta}(t + w_i^{(k)}))\}. \tag{4.3.6}$$

Note that, we can check all modes $k = 1, 2, \ldots, m_i$ for rectangle i at this stage. It takes $O(\tau)$ time to compute $p_{\max}(\pi)$ for each mode.

Now we evaluate the time complexity of ESM. It first takes $O(\tau n \log n)$ time to compute an optimal packing $\tilde{\pi}^* \in \Pi_{\tilde{\sigma},\tilde{\mu}}$ with algorithm CMPF. Time to compute $\tilde{f}_{\alpha,\beta}(x)$ and $\tilde{b}_{\alpha,\beta}(x)$

for all pairs of α and β by (4.3.4) and (4.3.5) is $O(\tau n^2)$. Time to compute the minimum penalty value by (4.3.6) is $O(\tau)$ for each pair of α and β, and it becomes $O(\tau n^2)$ for all pairs of α and β. In summary, the total computational time of this algorithm, i.e., time to evaluate all solutions when rectangle i is shifted in the shift neighborhood, is $O(\tau n^2)$. That is, it takes $O(\tau)$ amortized time to evaluate a coded solution in the neighborhood, and this becomes $O(1)$ for special cases in which $\tau = O(1)$ (e.g., area minimization, strip packing, and two-dimensional knapsack).

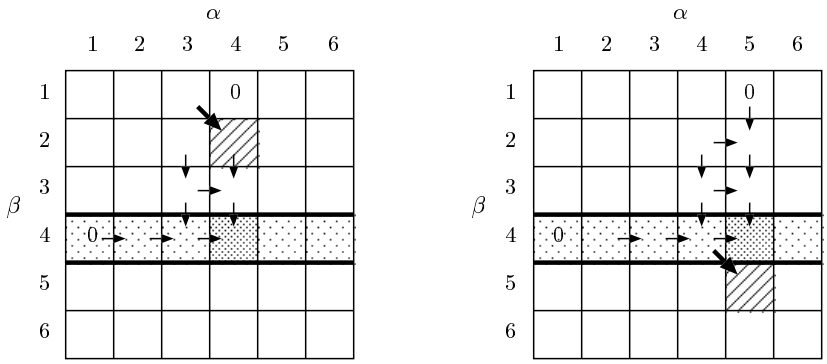
4.3.2 Evaluating limited shift moves

We propose an algorithm to evaluate solutions in the limited shift neighborhood where rectangle i will be shifted. It is clear that we can evaluate all solutions in this neighborhood with the previous algorithm, however, taking $O(\tau n)$ amortized time to evaluate each coded solution, since there are only $O(n)$ solutions in this case. Therefore, we introduce some ideas to compute $\tilde{f}_{\alpha,\beta}(x)$ and $\tilde{b}_{\alpha,\beta}(x)$ only for those α, β necessary to evaluate solutions in this neighborhood. Note that, the limited shift neighborhood is based on three kinds of operations defined in Subsection 4.2.2, and we propose ideas for each operation.

To evaluate solutions obtainable by a limited double shift operation, we can obtain $\tilde{f}_{\alpha,\beta}(x)$ from the computation of algorithm CMPF for $\tilde{\pi}^*$. That is, if we insert rectangle i before or after rectangle $j = \tilde{\sigma}_+(\alpha - 1) = \tilde{\sigma}_-(\beta - 1)$ in both permutations, we use $\tilde{f}_{\alpha-1,\beta-1}(x)$, $\tilde{f}_{\alpha-1,\beta}(x)$, $\tilde{f}_{\alpha,\beta-1}(x)$ or $\tilde{f}_{\alpha,\beta}(x)$ to evaluate a new coded solution (e.g., if i is inserted 'after' j in both $\tilde{\sigma}_+$ and $\tilde{\sigma}_-$, $\tilde{f}_{\alpha,\beta}(x)$ is necessary to compute (4.3.6)). Let us define $\tilde{J}^{\mathrm{f}}_j$ and $\tilde{f}_j(x)$ in a similar way as J^{f}_i and $f_i(x)$. Function $\tilde{f}_j(x)$ is computed in the computation CMPF for $\tilde{\pi}^*$. Then, $\tilde{f}_{\alpha,\beta}(x)$ is equal to $\tilde{f}_j(x)$, since $\tilde{J}^{\mathrm{f}}_{\alpha,\beta} = \tilde{J}^{\mathrm{f}}_j \cup \{j\}$. Moreover, other functions $\tilde{f}_{\alpha-1,\beta-1}(x)$, $\tilde{f}_{\alpha-1,\beta}(x)$ and $\tilde{f}_{\alpha,\beta-1}(x)$ are equal to $\max_{j' \in \tilde{J}^{\mathrm{f}}_j} \tilde{f}_{j'}(x)$, which is a part of (4.3.2) and has been already computed, since $\tilde{J}^{\mathrm{f}}_{\alpha-1,\beta-1} = \tilde{J}^{\mathrm{f}}_{\alpha-1,\beta} = \tilde{J}^{\mathrm{f}}_{\alpha,\beta-1} = \tilde{J}^{\mathrm{f}}_j$. Therefore, we can compute $\tilde{f}_{\alpha,\beta}(x)$ for all necessary α and β in $O(\tau n \log n)$ time, which is the computational time of CMPF. We can compute $\tilde{b}_{\alpha,\beta}(x)$ for all necessary α and β in a similar way.

To evaluate solutions obtainable by a single shift operation where rectangle i will be shifted in σ_+, we should compute $\tilde{f}_{\alpha,\beta}(x)$ for all $1 \leq \alpha \leq n$ and $\beta = \sigma_-^{-1}(i)$. $\tilde{f}_{\alpha,\beta}(x)$ can be computed for all necessary α and β by

$$\tilde{f}_{\alpha,\beta}(x) = \begin{cases} \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{j'}(x)\}, & \text{if } \tilde{\sigma}_-^{-1}(j') \leq \beta - 2, \\ \tilde{f}_{j'}(x), & \text{if } \tilde{\sigma}_-^{-1}(j') = \beta - 1, \\ \tilde{f}_{\alpha-1,\beta}(x), & \text{otherwise}, \end{cases} \tag{4.3.7}$$



(a) $\tilde{\sigma}_-^{-1}(\tilde{\sigma}_+(\alpha-1)) \le \beta-2$ (b) $\tilde{\sigma}_-^{-1}(\tilde{\sigma}_+(\alpha-1)) \ge \beta$

Figure 4.3: An example of evaluating solutions obtainable by a single shift operation

where $j' = \tilde{\sigma}_+(\alpha-1)$. This is because $\tilde{f}_{\alpha,\beta}(x)$ defined by (4.3.4) satisfy (see Figure 4.3):

$$\begin{aligned}
\tilde{f}_{\alpha,\beta}(x) &= \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha,\beta-1}(x)\} = \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha-1,\beta-1}(x), \tilde{f}_{\alpha,\beta-2}(x)\} = \cdots \\
&= \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha-1,\beta-1}(x), \ldots, \tilde{f}_{\alpha-1,l+1}(x), \tilde{f}_{\alpha,l}(x)\} \\
&= \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{j'}(x)\}, \qquad \text{if } \tilde{\sigma}_-^{-1}(j') = l-1 \le \beta-2, \\
\tilde{f}_{\alpha,\beta}(x) &= \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha,\beta-1}(x)\} = \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha-1,\beta-1}(x), \tilde{f}_{\alpha,\beta-2}(x)\} = \cdots \\
&= \max\{\tilde{f}_{\alpha-1,\beta}(x), \tilde{f}_{\alpha-1,\beta-1}(x), \ldots, \tilde{f}_{\alpha-1,2}(x), \tilde{f}_{\alpha,1}(x)\} \\
&= \tilde{f}_{\alpha-1,\beta}(x), \qquad \text{if } \tilde{\sigma}_-^{-1}(j') \ge \beta.
\end{aligned}$$

Similarly, if i is shifted in σ_-, $\tilde{f}_{\alpha,\beta}(x)$ can be computed for all necessary α and β by

$$\tilde{f}_{\alpha,\beta}(x) = \begin{cases} \max\{\tilde{f}_{\alpha,\beta-1}(x), \tilde{f}_{j''}(x)\}, & \text{if } \tilde{\sigma}_+^{-1}(j'') \le \alpha-2, \\ \tilde{f}_{j''}(x), & \text{if } \tilde{\sigma}_+^{-1}(j'') = \alpha-1, \\ \tilde{f}_{\alpha,\beta-1}(x), & \text{otherwise,} \end{cases} \tag{4.3.8}$$

where $j'' = \tilde{\sigma}_-(\beta-1)$. Time to compute $\tilde{f}_{\alpha,\beta}(x)$ for all necessary α and β by (4.3.7) or (4.3.8) is $O(\tau n)$ if the information from algorithm CMPF is retained. The computation of $\tilde{b}_{\alpha,\beta}(x)$ is similar and is omitted.

To evaluate solutions obtainable by a near-place double shift operation where rectangle i will be shifted, we should compute $\tilde{f}_{\alpha,\beta}(x)$ for all $\alpha_l \le \alpha \le \alpha_u$ and $\beta_l \le \beta \le \beta_u$, where $\alpha_u - \alpha_l \le 2\lceil a\sqrt{n}\,\rceil$ and $\beta_u - \beta_l \le 2\lceil a\sqrt{n}\,\rceil$ hold (a is a parameter explained in Subsection 4.2.2). We first compute $\tilde{f}_{\alpha,\beta_l}(x)$ for $1 \le \alpha \le \alpha_u$ by (4.3.7) and $\tilde{f}_{\alpha_l,\beta}(x)$ for $1 \le \beta \le \beta_u$ by (4.3.8). Then, we use (4.3.4) to compute $\tilde{f}_{\alpha,\beta}(x)$ for all $\alpha_l+1 \le \alpha \le \alpha_u$ and $\beta_l+1 \le \beta \le \beta_u$ (see Figure 4.4 as an example). Therefore, we can compute $\tilde{f}_{\alpha,\beta}(x)$ for all necessary α and β in $O(\tau a^2 n)$ time. The computation of $\tilde{b}_{\alpha,\beta}(x)$ is similar and is omitted.

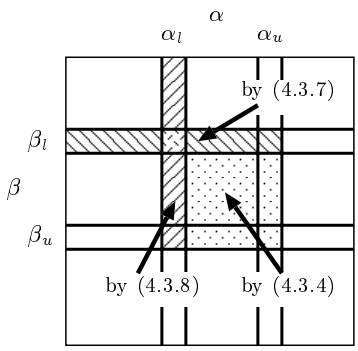


Figure 4.4: An example of evaluating solutions obtainable by a near-place double shift operation

In summary, we can evaluate all solutions in the limited shift neighborhood where rectangle i is shifted in $O(\tau n \log n)$ time. Therefore, amortized computational time to evaluate one coded solution becomes $O(\tau \log n)$. We call this algorithm Evaluate-Limited-Shift-Moves (ELSM). Note that we can evaluate solutions in the limited two-shifts neighborhood efficiently with similar ideas. The amortized computational time to evaluate one solution in the limited two-shifts neighborhood where rectangles i and j will be shifted is also $O(\tau \log n)$. This becomes $O(\log n)$ for special cases such as area minimization, strip packing and so on, since $\tau = O(1)$ holds for such cases as mentioned before.

4.4 Computational experiments

We conducted thorough computational experiments to evaluate the proposed algorithms. The algorithms were coded in the C language and run on a handmade PC (Intel Pentium III 1 GHz, 1 GB of memory). We used instances of three problems, whose data are obtainable electronically from `http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing/`.

4.4.1 Detailed comparisons of various CPUs

We first compare various CPUs in order to compare the performance of various algorithms, ILS-CBP, SA-BSG, SA-SP, BLF, SA-BLF, QH, TS-RPGSC and our ILS-ELSM, as fairly as possible (see the subsequent subsections for each algorithm). Table 4.1 shows the benchmark results of SPECint2000 and SPECint95 from SPEC web page for related CPUs (`http://www.specbench.org/`). In case the data of a related CPU is not available, the data of similar CPUs are shown.

Based on these data, we compute rough estimates of the speed of the related CPUs and show them in Table 4.1, where the speed of the Pentium III 1 GHz (CPU for ILS-CBP,

Table 4.1: Speed of various CPUs

CPU (algorithms)	SPECint2000	SPECint95	estimated speed
Pentium III 1 GHz	418		1
(ILS-CBP, ILS-ELSM, TS-RPGSC)			
Pentium III 933 MHz	398		0.97
Pentium III 910 MHz			0.95
(SA-BSG, SA-SP)			
Pentium III 866 MHz	380		0.91
Pentium III 800 MHz	361	38.7	0.86
Pentium Pro 200 MHz		8.08	0.18
(BLF, SA-BLF)			
Sun Sparc 20/71		3.11	0.07
(QH)			

ILS-ELSM and TS-RPGSC) is normalized as one, and a larger value means the speed is faster.

4.4.2 Comparison with our previous algorithms

We conducted two kinds of computational experiments to compare the proposed algorithms with our algorithms in Chapter 3.

Decoding algorithms

We examine the performance of five decoding algorithms: (1) an $O(\tau n^2)$ time naive algorithm in Subsection 3.3.1 (denoted NAIVE), (2) algorithm CMPF in Subsection 3.3.1 whose time complexity is $O(\tau n \log n)$, (3) algorithm CBP in Subsection 3.3.4 whose time complexity is $O(\tau n \log n)$, (4) algorithm ESM in Subsection 4.3.1 whose amortized time complexity is $O(\tau)$, and (5) algorithm ELSM in Subsection 4.3.2 whose amortized time complexity is $O(\tau \log n)$. All time complexities are for one coded solution. We tested instances of several sizes and several cost functions. Results are shown in Table 4.2. The figures in the table show the computational time to evaluate one 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 some special cost functions and then τ is $O(1)$, and "general" means that rectangles have general cost functions, each with a few segments, and then τ is $O(n)$.

Table 4.2: Computational time of the decoding algorithms in seconds

n	cost type	NAIVE	CMPF	CBP	ESM	ELSM
49	special	1.01×10^{-4}	3.80×10^{-5}	8.31×10^{-5}	2.85×10^{-7}	7.18×10^{-7}
100	special	4.42×10^{-4}	9.12×10^{-5}	2.01×10^{-4}	4.26×10^{-7}	6.91×10^{-7}
146	special	9.12×10^{-4}	1.38×10^{-4}	3.03×10^{-4}	4.96×10^{-7}	8.47×10^{-7}
200	special	1.93×10^{-3}	1.92×10^{-4}	4.23×10^{-4}	5.55×10^{-7}	7.84×10^{-7}
500	special	1.13×10^{-2}	7.30×10^{-4}	1.58×10^{-3}	5.39×10^{-7}	1.20×10^{-6}
49	general	9.06×10^{-4}	3.70×10^{-4}	3.27×10^{-4}	7.66×10^{-6}	9.81×10^{-6}
100	general	4.22×10^{-3}	8.66×10^{-4}	7.79×10^{-4}	7.44×10^{-6}	1.14×10^{-5}
146	general	8.48×10^{-3}	1.90×10^{-3}	1.54×10^{-3}	7.50×10^{-6}	1.42×10^{-5}
200	general	2.06×10^{-2}	2.54×10^{-3}	2.16×10^{-3}	7.49×10^{-6}	1.37×10^{-5}
500	general	1.19×10^{-1}	1.23×10^{-2}	1.08×10^{-2}	8.44×10^{-6}	1.90×10^{-5}

From Table 4.2, we can observe that proposed algorithms ESM and ELSM are much more efficient than previous algorithms even for small instances such as $n = 49$.

Local search and metaheuristic algorithms

We compare the performance of the proposed iterated local search algorithm (denoted ILS-ELSM), in which proposed decoding algorithms are incorporated, with our previous ILS algorithm (denoted ILS-CBP). Note that ILS-CBP is based on a decoding algorithm CBP, which was observed to be superior to another ILS algorithm based on CMPF if the same neighborhoods are used (see Subsection 3.5.2). ILS-CBP uses standard neighborhoods (swap, shift and change mode) and swap* neighborhood, where swap* is a subset of the two-shifts neighborhood. ILS-ELSM uses the combination of limited shift, limited two-shifts and change mode neighborhoods, and evaluates solutions with ELSM. We tested instances of the area minimization problem with up to 500 rectangles. We terminate the search when a prespecified computational time is reached even if we have not found any locally optimal solution, and output the best solution obtained during the search. Results are shown in Table 4.3. Column "value" shows the average of the following ratio,

$$100 \times \frac{\text{(the lower bound of the objective function)}}{\text{(the objective value of the output optimum)}},$$

for ten trials (i.e., the larger the better). Column "time" shows the time limit (in seconds) for one instance. These notations are also used in Tables 4.4 and 4.6. Column "lopt" shows the average number of obtained locally optimal solutions (i.e., "0.0" means each

Table 4.3: Comparison with our previous algorithm for area minimization

	ILS-CBP				ILS-ELSM			
instance	value	time	lopt	move	value	time	lopt	move
ami49	96.30	100.0	52.7	2987.5	97.37	100.0	61.9	5064.8
rp100	95.76	200.0	8.1	1176.5	96.78	200.0	15.7	2526.4
pcb146	95.63	300.0	11.8	1305.8	96.71	300.0	18.5	2347.2
rp200	95.67	400.0	0.3	495.3	96.30	400.0	3.3	1249.4
pcb500	92.27	1000.0	0.0	457.8	96.28	1000.0	0.4	800.0

run is forced to stop by the time limit before any locally optimal solution is found). Column "move" shows the average number of moves from a solution to a better solution. From the table, we can observe that ILS-ELSM is superior to ILS-CBP in quality for all instances. The numbers of obtained locally optimal solutions and moves by the proposed ILS algorithm are not so large, even though the time to evaluate a solution is much smaller than ILS-CBP, since the size of the neighborhood we use is larger than the previous one.

4.4.3 Comparison with other existing algorithms for area minimization

In this subsection, we compare our algorithm ILS-ELSM with two existing algorithms for the area minimizing problem: (1) A simulated annealing algorithm with the BSG coding scheme by Nakatake et al. [92] (denoted SA-BSG) and (2) a simulated annealing algorithm with the sequence pair coding scheme by Murata et al. [90] (denoted SA-SP). Note that the definition of this problem and the reduction to RPGSC form were given in Subsection 2.4.2, and test instances of this problem were explained in Subsection 3.5.1.

Algorithms SA-BSG and SA-SP were coded in the C language and run on a PC (Intel Pentium III 910 MHz), where the speed of this CPU is similar to ours (slightly slower, see Subsection 4.4.1 for more detailed comparison). These algorithms are specially tailored to the rectangle packing problem of minimizing the area, and their results for rp200 are not available (denoted N.A.). All results are shown in Table 4.4. Notations of this table are the same as Table 4.3. From Table 4.4, we can observe that ILS-ELSM is superior to SA-SP in both of the solution quality and the computational time for all instances. We can also observe that ILS-ELSM is superior to SA-BSG for almost all instances, and this tendency becomes clearer for larger instances. Note that our algorithm is designed to solve more general problems. Taking the generality of ILS-ELSM into consideration, the performance of ILS-ELSM appears to be quite satisfactory.

Table 4.4: Comparison with other methods for the area minimization problem

	SA-BSG		SA-SP		ILS-ELSM	
instance	value	time	value	time	value	time
ami49	97.10	69.0	96.29	176.0	97.29	65.0
rp100	97.08	68.2	88.54	248.7	96.40	65.0
pcb146	94.87	100.2	94.42	678.7	96.16	100.0
rp200	N.A.	N.A.	N.A.	N.A.	95.75	150.0
pcb500	94.10	334.6	90.82	7802.9	95.47	300.0

4.4.4 Comparison with other existing algorithms for strip packing

In this subsection, we compare our algorithm with other algorithms for the strip packing problem. Although this problem has been defined in Chapter 2, we explain the problem again and show test instances for our computational experiments. Then we show computational results for these instances.

Definition of the strip packing problem

We are given a set of n rectangles $I = \{1, 2, \ldots, n\}$ and the width W of a large object (strip), 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 height of the strip that contains all the given rectangles.

In our formulation as RPGSC instances, each rectangle has two modes of orientations. For $i = 1, 2, \ldots, n$, we set the width and height

$$w_i^{(1)} = w_i, \qquad h_i^{(1)} = h_i,$$
$$w_i^{(2)} = h_i, \qquad h_i^{(2)} = w_i.$$

For each $i = 1, 2, \ldots, n$ and $k = 1, 2$, we set cost functions

$$p_i^{(k)}(x) = \begin{cases} +\infty & (x < 0) \\ 0 & (0 \le x \le W - w_i^{(k)}) \\ \alpha(x - W + w_i^{(k)}) + \beta & (x > W - w_i^{(k)}), \end{cases} \tag{4.4.9}$$

$$q_i^{(k)}(y) = \begin{cases} +\infty & (y < 0) \\ y + h_i^{(k)} & (y \ge 0), \end{cases} \tag{4.4.10}$$

where α and β are nonnegative constants such that at least one of them is positive. We set $\alpha = 10$ and $\beta = 0$. The objective of the resulting RPGSC instance is to minimize

Table 4.5: Test instances of the strip packing problem

category	#rectangles n	bin width W	optimal height
C1	16 or 17	20	20
C2	25	40	15
C3	28 or 29	60	30
C4	49	60	60
C5	73	60	90
C6	97	80	120
C7	196 or 197	160	240

$p_{\max}(\pi) + q_{\max}(\pi)$ (i.e., the height of the strip if $p_{\max}(\pi) = 0$). We use test instances given by Hopper and Turton [58]. There are seven different categories called C1, C2, ..., C7 with the number of rectangles ranging from 16 to 197, where each category has three instances. The optimal value is known for all categories (see Table 4.5).

Computational results

We compared our algorithm ILS-ELSM with three existing heuristic algorithms for the strip packing problem: (1) A heuristic algorithm *bottom left fill with decreasing width* proposed by Chazelle [19] and implemented by Hopper and Turton [58] (denoted BLF), (2) a simulated annealing algorithm with BLF algorithm by Hopper and Turton [58] (denoted SA-BLF) and (3) an effective quasi-human based heuristic by Wu et al. [116] (denoted QH). The results of algorithms BLF and SA-BLF are taken from [58], where these algorithms were coded in the C++ language and run on a PC (Intel Pentium Pro 200 MHz, 65 MB memory). The results of algorithm QH are taken from [116], where algorithm QH was run on a SUN Sparc 20/71 (71 MHz SuperSparc CPU, 64 MB memory). Based on the benchmark results of SPECint, our CPU is about six times faster than Intel Pentium Pro 200 MHz and fourteen times faster than SuperSparc 71 MHz (see Subsection 4.4.1 for more detailed comparisons). Note that QH is not designed for solving the strip packing problem but for the rectangle packing problem with a bounding box. Therefore, to solve the strip packing problem with QH, Wu et al. run their algorithm many times while increasing the size of the bounding box by one unit per iteration (not only the height but the width of large object may be increased) to find the minimum bounding box in which all rectangles can be packed. As they change W, the problem solved by QH is different from others.

Results are shown in Table 4.6. Notations of this table are the same as Table 4.3,

Table 4.6: Comparison with other methods for the strip packing problem

	BLF		SA-BLF		QH†		ILS-ELSM	
category	value	time	value	time	value	time	value	time
C1	89	< 0.1	96	42	95.24	1.63	97.56	10.0
C2	84	< 0.1	94	144	97.92	6.19	93.75	15.0
C3	88	< 0.1	95	240	96.77	17.17	96.67	20.0
C4	95	< 0.1	97	1980	97.29	221.3	96.88	150.0
C5	95	< 0.1	97	6900	98.36	905.3	97.02	500.0
C6	95	< 0.1	97	22920	98.36	4581	96.85	1000.0
C7	95	0.64	96	250860	N.A.	N.A.	96.55	3600.0

†QH may increase the width of large object not only the height.

except for the number of trials; the results are the average of ten trials for SA-BLF and ILS-ELSM, however, only one trial for BLF and QH, since these two algorithms are deterministic. Note that the result of QH for C7 is not available (denoted N.A.) and their computational time does not include the time of the trial iterations to find the minimum bounding box in which all rectangles can be packed with QH (i.e., the time is reported only for the "successful" iteration). From Table 4.6, we can observe that BLF is much faster than other algorithms ("< 0.1" in this table means the computational time is less than 0.1 seconds), but the quality of solutions is a little worse than other algorithms. ILS-ELSM is slightly superior to SA-BLF in terms of the solution quality with less computational time. It is not easy to compare ILS-ELSM with QH, since the problem solved by QH is different from the strip packing problem. The solution quality of QH is quite good, however, ILS-ELSM seems very effective since it can deal with large instances such as $n = 196$ or more, and does not need any preliminary adjustments.

Now, we show the detailed results of our algorithm ILS-ELSM on the Hopper and Turton's instances [58]. There are seven different categories called C1, C2, ..., C7 and each category has three instances called P1, P2 and P3. We ran our algorithm ILS-ELSM ten times for each instance, and results are shown in Table 4.7. Column "average" (resp., "best" and "worst") shows the average (resp., best and worst) of the following ratio,

$$100 \times \frac{\text{(the optimal value)}}{\text{(the objective value of the output solution)}},$$

for ten trials. We terminated the search with time limit (specified in column "time"), and column "lopt" shows the average number of obtained locally optimal solutions.

Table 4.7: Detailed results by ILS-ELSM for Hopper and Turton's instances

instance	n	average	best	worst	time	lopt
C1-P1	16	99.50	100.0	95.24	10.0	411.0
C1-P2	17	95.69	100.0	95.24	10.0	297.6
C1-P3	16	97.56	100.0	95.24	10.0	322.4
C2-P1	25	93.75	93.75	93.75	15.0	155.5
C2-P2	25	93.75	93.75	93.75	15.0	167.6
C2-P3	25	93.75	93.75	93.75	15.0	195.9
C3-P1	28	96.77	96.77	96.77	20.0	194.3
C3-P2	29	96.77	96.77	96.77	20.0	220.3
C3-P3	28	96.46	96.77	93.75	20.0	224.6
C4-P1	49	96.93	98.36	96.77	150.0	233.0
C4-P2	49	96.77	96.77	96.77	150.0	268.2
C4-P3	49	96.93	98.36	96.77	150.0	265.6
C5-P1	73	96.98	97.83	96.77	500.0	199.8
C5-P2	73	96.77	97.83	95.74	500.0	221.8
C5-P3	73	97.30	98.90	96.77	500.0	195.7
C6-P1	97	96.93	97.56	96.77	1000.0	158.1
C6-P2	97	96.46	96.77	96.00	1000.0	200.3
C6-P3	97	97.16	97.56	96.77	1000.0	181.6
C7-P1	196	96.42	96.77	96.00	3600.0	54.5
C7-P2	197	96.70	97.17	96.00	3600.0	62.4
C7-P3	196	96.54	96.77	96.00	3600.0	64.7

Table 4.8: Comparison with other methods for the scheduling problem

	TS-RPGSC		ILS-CBP		ILS-ELSM	
instance	ratio	time	ratio	time	ratio	time
sp50-a	10/10	394.8	10/10	44.98	10/10	41.63
sp50-b	1/10	2730.6	10/10	125.0	10/10	47.60
sp78	10/10	427.2	10/10	405.6	7/10	1680.4
sp100-a	10/10	1851.9	10/10	362.6	3/10	1954.5
sp100-b	9/10	1230.1	10/10	47.03	10/10	574.9

4.4.5 Comparison with other algorithms on scheduling problems

In this subsection, we compared the performance of our algorithm with a tabu search algorithm developed for the resource constrained project scheduling problem by Nonobe and Ibaraki [93] (denoted TS-RPGSC) and our previous algorithm ILS-CBP on the scheduling problem explained in Subsection 3.5.1. As we mentioned in the previous chapter, to solve the problem by TS-RPGSC, we reformulated the problem as a project scheduling problem in the way as described in [56]. Note that TS-RPGSC 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 terminate 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. Results are shown in Table 4.8. Column "ratio" shows (the number of successful trials) / (the number of trials). Column "time" shows the average computational time (in seconds) to find an optimal solution in successful trials. From Table 4.8, we can observe that ILS-ELSM is superior to TS-RPGSC and ILS-CBP for some instances such as sp50-a and sp50-b, but ILS-CBP seems to be the best among the three algorithms. By preliminary experiments, an iterated local search algorithm, which is similar to ILS-CBP but decoding algorithm CMPF is incorporated, is inferior to ILS-CBP and ILS-ELSM. We therefore conclude that decoding algorithm CBP works well for this scheduling problem. It is our future work to propose another improved decoding algorithm maintaining good characteristics of CBP and ELSM.

4.5 Conclusion

In this chapter, we tackled the rectangle packing problem with general spatial costs, which contains numerous 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 speed-up techniques to evaluate solutions in various neighborhoods for this problem. By ESM (resp., ELSM), we realized $O(\tau)$ (resp., $O(\tau \log n)$) amortized computational time to evaluate one coded solution, where τ is the space complexity of the minimum penalty function. It becomes $O(1)$ or $O(\log n)$ for important special cases such as the area minimization, strip packing, two-dimensional knapsack and so on.

We also reported computational results for two variants of the rectangle packing problem, area minimization and strip packing, and a real-world scheduling problem. The computational results were quite encouraging for our algorithm proposed in this chapter.

Chapter 5

Local Search Algorithms for 2DCSP with a Given Number of Different Patterns

5.1 Introduction

In this chapter[1], we consider the two-dimensional cutting stock problem. As we mentioned in Chapter 2, it is one of the representative combinatorial optimization problems, and arises in many industries such as steel, paper, wood, glass and fiber. The problem can be defined as follows: We are given a sufficient number of stock sheets of the same width W and height H, and n types of rectangular products, where each product i has its width w_i, height h_i and demand d_i. From stock sheets we have to cut rectangular products, whose number is specified as demands. The objective is to minimize the total number of stock sheets required. This problem is NP-hard, since this is a generalization of the two-dimensional bin packing problem and the (one-dimensional) cutting stock problem, which are already known to be NP-hard [39].

As we mentioned in Subsection 1.2.1, it is often impractical to use many different cutting patterns in recent cutting industries, and some researchers (e.g., Foerster and Wäscher [37], Haessler [54], Umetani et al. [108]) have proposed algorithms for the one-dimensional cutting stock problem with consideration on the number of cutting patterns. In this chapter, we consider the two-dimensional cutting stock problem using a given number of different patterns m (we call this problem 2DCSPm). 2DCSPm asks to determine

[1]The results of this chapter appear in: S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two dimensional cutting stock problem," *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics* Volume IX (2003) 334–339, [65, 66].

a set of cutting patterns, whose size is m or less, and the numbers of applications of each cutting pattern. The objective is to minimize the total number of applications of cutting patterns.

The problem of deciding the number of applications for each pattern becomes an integer programming problem (IP). In Section 5.3, we propose a heuristic algorithm for the IP, which is based on its linear programming (LP) relaxation. We incorporate a sensitive analysis technique and the criss-cross method [119], a variant of simplex method, into our algorithm for efficiency. In Section 5.4, we propose local search algorithms to find a good set of cutting patterns. As the size of the neighborhood plays a crucial role in determining the efficiency of local search, we propose to utilize the dual solution of the LP relaxation for the purpose of restricting the number of solutions in the neighborhood. We design two kinds of neighborhood, basic and enhanced. Then they are computationally compared from the view point of the performance of the resulting local search algorithms.

To generate a feasible cutting pattern, we have to place all the given products in the stock sheet (two-dimensional area) without mutual overlap. At this placement stage, we assume that each product can be rotated by 90°, and assume no constraint on products' placement such as "guillotine cut". In Section 5.5, we first propose simple methods to check the feasibility of a given set of products. That is, under some conditions, we could easily find a feasible placement for a given pattern quickly or its infeasibility. In general, however, the problem to place all the products in a stock sheet without mutual overlap is NP-hard. In order to find a feasible placement, we use a local search algorithm for RPGSC with the sequence pair coding scheme [90] proposed in Chapters 3 and 4. It is however computationally too expensive if we always use the original rectangle packing algorithm, since we must solve the problem many times. We therefore modify it to a faster heuristic algorithm.

In Section 5.7, we generate random test instances of 2DCSP and conduct computational experiments to compare our algorithms with various different neighborhood operations. We also compute the trade-off curves between the number of different cutting patterns m and the solution quality.

5.2 Problem

5.2.1 Formulation

To define the two-dimensional cutting stock problem (2DCSP), we are given a sufficient number of stock sheets of the same width W and height H, and n types of rectangular products $I = \{1, 2, \ldots, n\}$, where each product i has its width w_i, height h_i and demand d_i.

A cutting pattern p_j is described as $p_j = (a_{1j}, a_{2j}, \ldots, a_{nj})$, where $a_{ij} \in \boldsymbol{Z}_+$ is the number of product i cut from a stock sheet by pattern p_j. A placement of products in a pattern is a set of their locations in one stock sheet together with their orientations (i.e., the original direction or rotated by 90°), where a placement is feasible if all the products are placed in one stock sheet without mutual overlap. We call a pattern p_j feasible if it has a feasible placement. Let S denote the set of all feasible patterns. Note that, the set S is very large and it is not explicitly given; i.e., we must find a feasible placement to confirm that a pattern is feasible.

A solution of 2DCSP consists of (1) a set of cutting patterns $\Pi = \{p_1, p_2, \ldots, p_{|\Pi|}\} \subseteq S$, (2) a feasible placement of each pattern $p_j \in \Pi$, and (3) the numbers of applications $X = (x_1, x_2, \ldots, x_{|\Pi|})$ of all the patterns $p_j \in \Pi$, where $x_j \in \boldsymbol{Z}_+$. A typical cost function is the total number of stock sheets used in a solution. This problem is formally described as follows:

$$\begin{aligned} \text{2DCSP:} \quad & \text{minimize} && f(\Pi, X) = \sum_{p_j \in \Pi} x_j \\ & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in I, \\ & && \Pi \subseteq S, \\ & && x_j \in \boldsymbol{Z}_+, \text{ for } p_j \in \Pi. \end{aligned} \tag{5.2.1}$$

Here we consider a variant of 2DCSP with an input parameter m, where m is the number of different cutting patterns $|\Pi|$. We call this problem the two-dimensional cutting stock problem with a given number of different patterns m (2DCSPm), which is formally defined as follows:

$$\begin{aligned} \text{2DCSP}m\text{:} \quad & \text{minimize} && f(\Pi, X) = \sum_{p_j \in \Pi} x_j \\ & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in I, \\ & && \Pi \subseteq S, \\ & && |\Pi| \leq m, \\ & && x_j \in \boldsymbol{Z}_+, \text{ for } p_j \in \Pi. \end{aligned} \tag{5.2.2}$$

5.2.2 Lower bound

In this subsection, we consider lower bounds of the total number of stock sheets used for 2DCSP. A simple lower bound is the so-called *continuous lower bound*, L_1, which is defined

as follows:

$$L_1 = \left\lceil \sum_{i \in I} d_i w_i h_i / WH \right\rceil . \tag{5.2.3}$$

This bound is easy to compute (it can be computed in $O(n)$ time), and is a good bound when there are many products of small sizes. We also introduce another lower bound L_2, which works effective if there are many large products. This lower bound is obtained by concentrating on large products, and is a little complicated since each product can be rotated by 90°. We note that various lower bounds are known for the two-dimensional bin packing problem without rotation [84].

Now for each product i, we define w_i^* and h_i^* as follows:

$$\begin{aligned} &w_i^* = h_i^* = \min\{w_i, h_i\} && (\text{if } \max\{w_i, h_i\} \le \min\{W, H\}), \\ &w_i^* = w_i,\ h_i^* = h_i && (\text{if } w_i > H \text{ or } h_i > W), \\ &w_i^* = h_i,\ h_i^* = w_i && (\text{if } w_i > W \text{ or } h_i > H). \end{aligned} \tag{5.2.4}$$

Given a constant q, with $0 < q \le H/2$, we define

$$\begin{aligned} &I^W = \{i \in I : w_i^* > W/2\}, \\ &I_1^W(q) = \{i \in I^W : h_i^* > H - q\}, \\ &I_2^W(q) = \{i \in I^W : H - q \ge h_i^* \ge q\}. \end{aligned} \tag{5.2.5}$$

No two products $i \in I_1^W(q)$ and $j \in \{I_1^W(q) \cup I_2^W(q)\}$ can be packed into the same stock sheet, and at most $\lfloor H/q \rfloor$ products in $I_2^W(q)$ can be packed into one stock sheet. Based on this, we have the following lower bound:

$$L_2^W = \max_{0<q\le H/2} \left(\sum_{i \in I_1^W(q)} d_i + \left\lceil \frac{\sum_{i \in I_2^W(q)} d_i}{\lfloor H/q \rfloor} \right\rceil \right). \tag{5.2.6}$$

This bound can be computed in $O(n \log n)$ time with the following algorithm.

Algorithm: Compute L_2^W

Step 1: Divide all products into two sets I^W and $I \setminus I^W$. For each product $i \in I^W$, if $h_i^* \le H/2$, set $s(i) := h_i^*$; otherwise set $s(i) := H - h_i^*$. Sort rectangles $i \in I^W$ in the ascending order of $s(i)$. Set $L_2^W := 0, q := 0$, $I_1^W(q) := \emptyset$ and $I_2^W(q) := I^W$.

Step 2: Choose a rectangle $i \in I_2^W(q)$ with the smallest $s(i)$. (If $s(i) = s(j)$ and $h_i^* > h_j^*$ hold, we choose i before j.) If $h_i^* \le H/2$, go to Step 3; Otherwise go to Step 4.

Step 3: Set $q := h_i^*$ and compute $\sum_{i\in I_1^W(q)} d_i + \left\lceil \sum_{i\in I_2^W(q)} d_i \;/\; \lfloor H/q \rfloor \right\rceil$. If this is larger than L_2^W, set $L_2^W := \sum_{i\in I_1^W(q)} d_i + \left\lceil \sum_{i\in I_2^W(q)} d_i \;/\; \lfloor H/q \rfloor \right\rceil$. Set $I_2^W(q) := I_2^W(q) \setminus \{i\}$. If $I_2^W(q) = \emptyset$, output L_2^W and halt; otherwise return to Step 2.

Step 4: Set $q := H - h_i^* + \varepsilon$, $I_1^W(q) := I_1^W(q) \cup \{i\}$ and $I_2^W(q) := I_2^W(q) \setminus \{i\}$, and compute $\sum_{i\in I_1^W(q)} d_i + \left\lceil \sum_{i\in I_2^W(q)} d_i \;/\; \lfloor H/q \rfloor \right\rceil$ (where ε is an arbitrarily small positive). If this is larger than L_2^W, set $L_2^W := \sum_{i\in I_1^W(q)} d_i + \left\lceil \sum_{i\in I_2^W(q)} d_i \;/\; \lfloor H/q \rfloor \right\rceil$. If $I_2^W(q) = \emptyset$, output L_2^W and halt; otherwise return to Step 2.

We also define $I^H, I_1^H(q)$ and $I_2^H(q)$ in the same manner by exchanging the roles of W and H, and another lower bound L_2^H is similarly computed. As a result, the following lower bound L_2 is derived by considering large products only.

$$L_2 = \max\left\{L_2^W, L_2^H\right\}. \tag{5.2.7}$$

It is easily seen that none of the L_1 and L_2 dominates the other. The overall lower bound we use in this paper is as follows,

$$f_{LB} = \max\left\{L_1, L_2\right\}. \tag{5.2.8}$$

5.3 Computing the number of pattern applications

In this section, we consider the problem of computing $X = (x_1, x_2, \ldots, x_m)$ for a given set of patterns $\Pi = \{p_1, p_2, \ldots, p_m\}$, where x_j denotes the number of applications of pattern p_j. This problem is known to be NP-hard, and it can be described as the following integer programming problem:

$$\begin{aligned} \mathrm{IP}(\Pi): \quad & \text{minimize} && f(X) = \sum_{p_j\in\Pi} x_j \\ & \text{subject to} && \sum_{p_j\in\Pi} a_{ij}x_j \geq d_i, \text{ for } i \in I, \\ & && x_j \in \boldsymbol{Z}_+, \text{ for } p_j \in \Pi. \end{aligned} \tag{5.3.9}$$

There are several studies for 2DCSP that solve this problem exactly with branch-and-bound method. However, as we must solve this problem many times in our algorithm, we consider a faster heuristic algorithm.

Our heuristic algorithm first solves the LP relaxation LP(Π) of IP(Π), in which the integer constraints $x_j \in \boldsymbol{Z}_+$ are replaced with $x_j \geq 0$.

$$\begin{aligned} \mathrm{LP}(\Pi): \quad & \text{minimize} && f(X) = \sum_{p_j \in \Pi} x_j \\ & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in I, \\ & && x_j \geq 0, \text{ for } p_j \in \Pi. \end{aligned} \tag{5.3.10}$$

Let $\overline{X} = (\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_m)$ denote an optimal solution of LP(Π). Although this solution is not integer valued, the gaps between the optimal values of IP(Π) and LP(Π) are observed to be small in most instances of 2DCSPm. Our algorithm is based on this observation. The simplest heuristic algorithm for this problem would be "rounding up"; i.e., we output $(\lceil \overline{x}_1 \rceil, \lceil \overline{x}_2 \rceil, \ldots, \lceil \overline{x}_m \rceil)$.

In our heuristic algorithm, we do a little more than the simple rounding up. We first set $\hat{x}_j := \lfloor \overline{x}_j \rfloor$ for all patterns $p_j \in \Pi$ and let $\hat{X} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m)$. We then sort all indices j in the descending order of $\overline{x}_j - \lfloor \overline{x}_j \rfloor$, and round up $\overline{x}_j$ to $\lceil \overline{x}_j \rceil$ or round down $\overline{x}_j$ to $\lfloor \overline{x}_j \rfloor$ in the resulting order of j according to the following rule. If pattern p_j includes a product i such that $\sum_j a_{ij} \hat{x}_j < d_i$, we set $\hat{x}_j := \hat{x}_j + 1$ and go to the next pattern. If pattern p_j does not include such products, we do not change $\hat{x}_j$ (i.e., rounding down) and go to the next. The solution $\hat{X}$ obtained by this process is always feasible to IP(Π) and not worse than the solution obtained by the simple rounding up method. Note that various rounding procedures are compared in e.g., Valdés et al. [110], Vanderbeck [111].

In our local search algorithm for finding a good set of patterns, which will be explained in the next section, we must solve many LP(Π). If LP is naively solved from scratch whenever we evaluate a new set of patterns, the computation becomes expensive. Therefore, as in the study of Umetani et al. [108], we incorporate a sensitive analysis technique and the criss-cross method [119], a variant of simplex method, to utilize an optimal LP solution for the current set of patterns Π. For each set of patterns $\Pi' \in N(\Pi)$, where $N(\Pi)$ is the family of neighborhood sets of patterns of Π, the criss-cross algorithm converges to an optimal solution of LP(Π') usually after a few pivot operations.

5.4 A local search algorithm to find a good set of patterns

In this section, we describe a local search (LS) procedure to find a good set of patterns $\Pi = \{p_1, p_2, \ldots, p_m\}$. It generates many sets of patterns Π' in the neighborhood $N(\Pi)$ of the current set of patterns Π. The numbers of applications of patterns in set Π' are computed by solving IP(Π') explained in the previous section.

The following ingredients must be specified in designing LS: Initial solution, neighborhood, move strategy and a function to evaluate solutions. In Subsection 5.4.1, we will explain how to prepare an initial feasible solution, and in Subsection 5.4.2, we will design several neighborhoods. As the move strategy, we adopt the first admissible move strategy (i.e., as soon as we find a better solution in its neighborhood, we move to the solution). A set of patterns Π is evaluated by the optimal value of LP relaxation LP(Π). Note that, we also compute an integer solution $\hat{X}$ of IP(Π) heuristically by the algorithm in Section 5.3, and update the incumbent solution accordingly (i.e., the best feasible solution among those obtained so far).

5.4.1 Initial solution

If there is no restriction on the number of different cutting patterns, it is easy to construct a feasible solution. However, just finding a feasible solution is not trivial for 2DCSPm, since it contains the two-dimensional bin packing problem as a subproblem. Hence, to design a local search algorithm for 2DCSPm, we first consider how to construct a feasible solution heuristically.

In order to construct a feasible solution, we ignore the demands d_i of all products $i \in I$, temporarily assuming them as one, and consider the following two-dimensional bin packing problem (2DBPP) instead of 2DCSPm:

$$\begin{aligned} \text{2DBPP:} \quad & \text{Find a set of patterns } \Pi, \\ & \text{subject to } \sum_{p_j \in \Pi} a_{ij} \geq 1, \text{ for } i \in I, \\ & \Pi \subseteq S, \\ & |\Pi| \leq m. \end{aligned} \tag{5.4.11}$$

If a feasible solution Π for 2DBPP is given, it is easy to construct a feasible solution for 2DCSPm by using appropriately large numbers of applications x_j of cutting patterns. We utilize a heuristic algorithm for 2DBPP which is based on the next-fit algorithm known for the one-dimensional bin packing problem. This algorithm has some kind of randomness, and it is possible to construct many different initial solutions. Note that, 2DBPP has been well studied and there are more sophisticated heuristic and metaheuristic algorithms [79].

5.4.2 Neighborhoods

As an example of natural neighborhoods, let $N(\Pi)$ be the set of solutions obtainable by replacing a cutting pattern p_j in the set Π with another cutting pattern $p'_j \in S \setminus \Pi$:

$$N(\Pi) = \left\{ \Pi \cup \{p'_j\} \setminus \{p_j\} \mid p_j \in \Pi, p'_j \in S \setminus \Pi \right\}, \tag{5.4.12}$$

where S is the set of all feasible cutting patterns. As mentioned in Subsection 5.2.1, the number of all feasible cutting patterns $|S|$ is too large, and most of them may not lead to improvement. In view of this, we propose heuristic algorithms to generate smaller neighborhoods.

Basic neighborhood

Let $(\Pi, \overline{X})$ be the current solution, where $\overline{X}$ is an optimal solution of LP(Π) which may not be integer valued. The family of sets of patterns in our reduced neighborhood of Π are those generated by changing one pattern $p_j \in \Pi$ by the following operation: Remove t ($t = 0, 1, 2$) products from p_j and add one product. We call this operation "basic operation" and the neighborhood defined by basic operations is called the basic neighborhood. Note that, "removing a product i from pattern p_j" means one unit of product i is removed from p_j (i.e., a_{ij} decreases by one). Adding a product is similarly defined. In order to decide which products to be removed, we use the overproduction

$$r_i = \sum_j a_{ij} x_j - d_i \tag{5.4.13}$$

of product i; we sort all products in the descending order of r_i, and then remove the products in this order. This is because it is impossible to improve the current solution by adding products i satisfying $r_i > 0$. On the other hand, in order to determine the product to be added, we use a dual optimal solution $\overline{Y} = (\overline{y}_1, \overline{y}_2, \ldots, \overline{y}_n)$ of LP(Π). Larger $\overline{y}_i$ indicates that increasing a_{ij} in pattern p_j is more effective. We sort all products satisfying $r_i = 0$ in the descending order of $\overline{y}_i$, and add a product in this order in a basic operation. The size of this neighborhood is $O(mn^{t+1})$ for a given t.

Redundancy reduction

To make the basic neighborhood more effective, we introduce other operations. For each cutting pattern p_j, products are divided into two sets. One is the set of products which do not affect the current LP solution even if one unit of the product is removed from p_j (i.e., the set of products i satisfying $a_{ij} \geq 1$ and $x_j \leq r_i$). The other is the set of products which affect the LP solution if it is removed from the pattern (i.e., the set of products i such that $a_{ij} \geq 1$ and $x_j > r_i$). The a_{ij} of each product i in the first set is reduced as much as possible as long as the current LP solution remains the same. We call this operation as the redundancy reduction operation, and it is applied before the basic operation.

Filling-up operation

We explain another operation of adding products after changing pattern p_j by the basic operation. We divide all products i into two sets according to whether overproduction r_i is 0 or positive. In this stage, we sort all products i with $r_i = 0$ in the descending order of $\overline{y}_i$, and add them one by one in this order as long as the resulting pattern is feasible. Whenever a product is added, we recompute an optimal solution of LP and update $\overline{y}_i$. When we exhaust all the products with $r_i = 0$ into pattern p_j, the products with $r_i > 0$ are added. In this stage, we sort such products in the ascending order of r_i, and add them in this order. Since products i with $r_i > 0$ do not affect the LP solution, we can not improve the current LP value by this operation. However, if we apply this operation, we may find a better solution in the subsequent iterations in our local search. We call this operation as the filling-up operation, and it is applied after the basic operation. By these two operations, we can improve the quality of pattern p_j.

Replacement operation

If all products in pattern p_j are removed by the redundancy reduction and basic operations, we must reconstruct a new pattern from scratch by the basic and filling-up operations. This situation always occurs for pattern p_j with $x_j = 0$, and such reconstruction may not find a pattern with small trim loss. We therefore replace p_j with a new pattern in this case. For this purpose, we keep cm (c is a parameter and we use $c = 3$) good cutting patterns obtained by then in memory, and choose one from them, where we define good cutting pattern as those having small trim loss. We call this as the replacement operation.

Enhanced neighborhood

Now, our new neighborhood is the set of solutions obtained from Π by applying the operations proposed in this section (i.e., basic, redundancy reduction, filling-up and replacement). We call this neighborhood as the enhanced neighborhood. Two neighborhoods, basic and enhanced, will be computationally compared in Section 5.7.

5.5 Feasibility check for a pattern

For a given cutting pattern p_j, we must check its feasibility (i.e., find a feasible placement of the products in the stock sheet). This feasibility check is trivial for the one-dimensional cutting stock problem, since we just check the following inequality:

$$\sum a_{ij} l_i \leq L, \tag{5.5.14}$$

where l_i is the length of product i and L is the length of a stock roll. On the other hand, the problem is hard for the two-dimensional case; because, in general, we must solve the two-dimensional rectangle packing problem.

In this section, we first propose two simple methods to check the feasibility of a given two-dimensional pattern. These methods, however, work well only for some special cases. We then propose a heuristic algorithm to find a feasible placement.

We first check a given pattern p_j against the continuous lower bound,

$$\sum_{i \in I} a_{ij} w_i h_i \leq WH. \tag{5.5.15}$$

The pattern is obviously infeasible if this inequality is not satisfied. The second method is used to check a pattern constructed in neighborhood search. In this case, we remove some products from the current pattern and add some products to the resulting pattern. A new pattern has a feasible placement if the products to be added are smaller than the removed products. For example, if we remove two products i and i' from a pattern, and then add two products, one of them is smaller than i and the other is smaller than i', then the resulting pattern is judged to be feasible without solving the rectangle packing problem.

In other cases, however, we must tackle the two-dimensional rectangle packing problem, which is NP-hard.

Rectangle Packing Problem

Input: A set of rectangles and one stock sheet, each of the rectangles and the stock sheet having its width and height.

Output: The locations (l_k^x, l_k^y) and orientations of all rectangles k in the stock sheet such that no two rectangles overlap each other.

There are many heuristic algorithms in the literature proposed for this problem. This problem can be reducible to RPGSC, and basically, we use the local search algorithm for RPGSC with the sequence pair coding scheme [90] proposed in Chapters 3 and 4. (We call this algorithm the original rectangle packing algorithm.) It is however computationally too expensive if we always use the original rectangle packing algorithm, since we must solve the problem many times. We therefore modify it to a faster heuristic algorithm in the following manner.

To construct an initial solution for the local search, we apply the original rectangle packing algorithm for each pattern in Π. In our neighborhood search, we modify a pattern $p_j \in \Pi$ to another pattern p'_j. Since we already have a good sequence pair σ for pattern p_j, we construct a sequence pair σ' for p'_j starting from σ. As noted in Subsection 5.4.2, we

remove some products from pattern p_j and add some other products to construct a new pattern p'_j. When a product i is removed, we just remove it from the current sequence pair σ. If p_j includes two or more units of product i (i.e., $a_{ij} \geq 2$), we remove one of them at random. When we add a new product i', we check the insertions of i' to all positions of the sequence pair. The number of sequence pairs we check in this process is s^2, where s is the number of products in pattern p_j (i.e., $s = \sum_i a_{ij}$). We check all of them in $O(s^2)$ time with the algorithm Evaluate-Shift-Moves proposed in Subsection 4.3.1. When there is more than one product to be added, we add them one by one to the sequence pairs generated by then.

5.6 The entire algorithm

In this section, we first describe algorithm NS$((\Pi, \overline{X}), (\Pi^*, \hat{X}^*), p_j, t)$, where NS stands for neighborhood search, and then describe the entire framework of our local search algorithm.

Algorithm NS$((\Pi, \overline{X}), (\Pi^*, \hat{X}^*), p_j, t)$ is the core routine of our algorithm, which is comprised of those algorithms described in Sections 5.3, 5.4 and 5.5. Starting from the current set of patterns Π, this algorithm checks the family of sets of patterns generated by modifying the pattern $p_j \in \Pi$ by the basic, redundancy reduction, filling-up and replacement operations, for a given parameter t ($= 0, 1$ or 2) used in the basic operation. It also computes an integer solution for each set of patterns by the heuristic algorithm of Section 5.3, and updates the incumbent solution $(\Pi^*, \hat{X}^*)$ if a better integer solution is obtained.

Algorithm: NS$((\Pi, \overline{X}), (\Pi^*, \hat{X}^*), p_j, t)$

Input: The current solution $(\Pi, \overline{X})$, the incumbent solution $(\Pi^*, \hat{X}^*)$, a pattern $p_j \in \Pi$ which is the candidate to be removed from Π, and a parameter $t = 0, 1$ or 2.

Output: An improved solution $(\Pi', \overline{X'})$ if it exists; otherwise 'no', while updating the incumbent solution $(\Pi^*, \hat{X}^*)$.

Step 1: If $\overline{x}_j = 0$, go to Step 2; otherwise go to Step 3.

Step 2: Repeat the following procedure cm times (where cm is the number of good cutting patterns in memory, as described in Subsection 5.4.2), and then go to Step 5.

(Replacement operation): Replace pattern p_j with a good cutting pattern stored in memory. Compute IP solution $(\Pi', \hat{X}')$ and LP solution $(\Pi', \overline{X'})$ for the resulting set of patterns as described

in Section 5.3. If $(\Pi', \hat{X}')$ is better than the incumbent solution $(\Pi^*, \hat{X}^*)$, update it. If $(\Pi', \overline{X'})$ is better than the current solution $(\Pi, \overline{X})$, exit with $(\Pi', \overline{X'})$.

Step 3 (redundancy reduction):

Remove the redundant products from p_j as many as possible, by the procedure in Subsection 5.4.2. Denote the resulting pattern as p_j^1. If this pattern does not have more than t products, go to Step 2; otherwise go to Step 4.

Step 4: Apply the following procedures to every subset of products I' with $|I'| = t$ and every product $i \notin I'$ with $r_i = 0$, as described in Subsection 5.4.2.

4-1 (basic operation):

Remove the products in I' from p_j^1 and add the product i. Denote the resulting pattern as p_j^2 and check its feasibility by the procedure described in Section 5. If a feasible placement is found, then compute the LP solution as described in Section 5.3, and go to 4-2.

4-2 (filling-up operation):

Fill-up the pattern p_j^2 by the procedure in Subsection 5.4.2. Compute IP solution $(\Pi', \hat{X}')$ and LP solution $(\Pi', \overline{X'})$ for the resulting set of patterns as described in Section 5.3. If $(\Pi', \hat{X}')$ is better than the incumbent solution $(\Pi^*, \hat{X}^*)$, update it. If $(\Pi', \overline{X'})$ is better than the current solution $(\Pi, \overline{X})$, exit with $(\Pi', \overline{X'})$.

Step 5: Exit with 'no' (no improved LP solution found).

Finally, the outline of our entire local search algorithm is described as follows.

Algorithm: LS_2DCSPm

Line 1: Construct an initial set of patterns Π and compute its LP solution $\overline{X}$;
Line 2: Set $\Pi^* := \Pi$ and compute its IP solution $\hat{X}^*$ (i.e., the incumbent solution);
Line 3: Start the neighborhood search from the current solution $(\Pi, \overline{X})$;
Line 4: **for** $t = 0, 1, 2$ **do**
Line 5: **for** $p_j \in \Pi$ **do**
Line 6: NS$((\Pi, \overline{X}), (\Pi^*, \hat{X}^*), t, p_j)$ to obtain an improved solution;
Line 7: **if** an improved solution $(\Pi', \overline{X'})$ is found **then**
Line 8: set $(\Pi, \overline{X}) := (\Pi', \overline{X'})$ and return to Line 3;
Line 9: **end for**

Line 10: **end for**
Line 11: Output the incumbent solution $(\Pi^*, \hat{X}^*)$ and halt;

5.7 Computational experiments

We conducted computational experiments to evaluate the proposed algorithms. The algorithms were coded in the C language and run on a handmade PC (Intel Pentium IV 2.8GHz, 1GB memory).

5.7.1 Other existing algorithms

In the literature, several heuristic algorithms have been proposed for 2DCSP [18, 24, 34, 109, 110, 112] and some of their computational results have been reported. For the evaluation of our algorithm, it is desirable to compare our algorithm with such computational results. However, it is not easy for the following reasons.

First of all, there are many variations of 2DCSP and those algorithms in the literature were designed for slightly different problems. Cung et al. [24] and Valdés et al. [109] considered the following problem, and proposed branch-and-bound and heuristic algorithms: Cut a single rectangular stock sheet into a set of small rectangular products of given sizes and values so as to maximize the total value. If the value of each product is equal to its area, the objective is to minimize the trim loss. Valdés et al. considered another problem in [110]: A set of stock sheets of different sizes and a set of rectangular products are given. Each product has its width, height, demand and a fixed orientation. From these stock sheets, products are cut by "guillotine cut" in order to satisfy all demands. The objective is to minimize the total area of stock sheets required. Vanderbeck [112] proposed a heuristic algorithm, based on a nested decomposition for 2DCSP with some constraints such as 3-stage pattern and the maximum number of products in one pattern. Chauny and Loulou [18] and Farley [34] considered a similar problem to ours, except that the number of different cutting patterns m is not specified. In [18] and [34], heuristic algorithms based on the column generation technique were proposed, together with some computational results. However, their computational results are too limited to compare.

From these observation, we had to give up the comparison with other existing algorithms. Instead, we generated various types of test instances, and conducted detailed experiments with two different types of neighborhoods and different numbers of patterns m.

5.7.2 Test instances

We generated random test instances of 2DCSP following the generation scheme described in [100, 110]. The instances are characterized by the following three parameters.

Number of product types: We have four classes 20, 30, 40 and 50 of the number of product types n (e.g., $n = 20$ in class 20).

Range of demands: Demand d_i of type S (S stands for small) is randomly taken from interval $[1, 25]$, type L (large) is taken from $[100, 200]$, and type V (variable) is taken from either intervals $[1, 25]$ or $[100, 200]$ with the equal probability for each product i.

Size of stock sheet: We have five classes $\alpha, \beta, \gamma, \delta$ and ε of the stock sheets. Class α is the smallest stock sheet which can contain six products on the average, while class ε is the largest containing about 50 products.

Hence, there are 60 types of instances and we generated one instance for each type. These instances are named like "20Sα", "20Sβ", ..., "20Sε", "20Lα", ..., "20Vε", "30Sα", ..., "50Vε". In our computational experiments, we apply our local search algorithms ten times to each instance with different initial solutions, and report the average results of ten trials. All test instances are electronically available from our web site (`http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing/`).

5.7.3 Comparison of basic and enhanced neighborhoods

First, basic and enhanced neighborhoods were computationally compared. For each instance, we applied our local search algorithm with each type of neighborhood ten times, and report the average quality of the obtained solutions and computational time, where local search halts only when a locally optimal solution is reached. For simplicity, we set the number of different cutting patterns to the number of product types (i.e., $m = n$). Results are shown in Table 5.1. Column "n" shows the number of product types. For each n, we have 15 instances with different ranges of demands and different sizes of stock sheet; e.g., we have instances 20Sα, 20Sβ, ..., 20Vε for $n = 20$. Column "quality" shows the average of the following ratio,

$$\text{quality} = 100 \cdot (f - f_{LB})/f_{LB}, \tag{5.7.16}$$

where f is the number of stock sheets used in the solution, and f_{LB} is a lower bound of the number of required stock sheets computed by (5.2.8) in Subsection 5.2.2. The smaller quality means the better performance of the algorithm. Column "time" shows

Table 5.1: Comparison two neighborhoods in solution quality and computational time

	basic		enhanced	
n	quality	time	quality	time
20	15.17	13.88	10.49	18.42
30	14.81	41.18	8.71	45.76
40	11.91	221.61	8.76	144.93
50	10.94	955.64	8.18	638.86

the average CPU time in seconds of one local search. These notations are also used in Tables 5.2 and 5.3.

From Table 5.1, we can observe that the enhanced neighborhood gives smaller quality value than the basic neighborhood in all cases, while using similar computational time. It indicates that the redundancy reduction, filling-up and replacement operations proposed in Subsection 5.4.2 make the search more powerful and efficient. Based on this, we will use the enhanced neighborhood in the following experiments.

5.7.4 Effect of the number of patterns m

Next, we conducted computational experiments for different number of cutting patterns m, i.e., m was set to $n, 0.8n, 0.6n$ and $0.4n$. Results are given in Table 5.2. (We also show the detailed results in Table 5.3.) The leftmost column of Table 5.2 shows the instance classes. For example, "class 20" represents 15 instances with $n = 20$. Each figure in this row is the average of 150 trials (that is, 10 trials with different initial solutions for each instance, and there are 15 instances for class 20). "class S" represents 20 instances whose demand is taken from interval $[1, 25]$, and each figure is the average of 200 trials. Other rows can be similarly interpreted. Now from the rows for classes 20, 30, 40 and 50 in Table 5.2, we can observe that as n becomes larger (i.e., from class 20 to 50), computational time increases and solution quality becomes slightly better. As m becomes smaller, the size of neighborhood becomes smaller and local search algorithm converges to locally optimal solutions rather quickly, making the quality of obtained solution poorer.

From the rows for different ranges of demands (i.e., S, L and V), we can observe that the solution quality for class S is the worst even though all classes use similar computational time. This is due to the influence of rounding and overproduction. Namely, we compute the numbers of applications x_j by rounding from the LP solution, and it introduces a little overproduction for several product types. As the total demands is smaller for class S, the

Table 5.2: Quality and time with various number of different patterns on various classes

	$m = n$		$m = 0.8n$		$m = 0.6n$		$m = 0.4n$	
	quality	time	quality	time	quality	time	quality	time
class 20	10.491	18.424	12.247	6.308	14.432	3.753	20.810	1.954
class 30	8.707	45.758	9.899	18.533	12.175	6.424	16.758	3.098
class 40	8.758	144.932	10.360	45.330	12.218	15.630	16.891	7.091
class 50	8.177	636.861	9.940	143.135	11.504	37.292	15.315	12.117
class S	12.531	149.200	14.555	47.335	16.377	15.373	20.769	6.348
class L	6.185	262.195	7.390	57.178	9.100	16.865	12.403	5.369
class V	8.384	223.087	9.890	55.467	12.269	15.086	19.158	6.478
class α	10.516	46.944	12.436	10.001	16.323	1.843	28.203	0.264
class β	8.048	71.712	10.027	15.579	12.269	3.388	19.436	0.780
class γ	7.924	141.175	9.661	28.295	11.530	6.258	14.912	1.627
class δ	7.331	311.113	8.483	60.768	9.397	15.827	10.795	6.567
class ε	11.348	486.525	12.451	151.990	13.391	51.559	13.871	21.087
average	9.033	211.494	10.612	53.327	12.582	15.775	17.443	6.065

effect of one unit of overproduction to the quality is larger.

From the rows for different sizes of stock sheet (i.e., class $\alpha, \beta, \gamma, \delta$ and ε), we can observe that the solution qualities for classes α and ε are worse than others. The reason for class ε is similar to the previous one. For many test instances of class ε, we could find good solutions if the numbers of applications x_j can be fractional. However, these solutions degrade after obtaining integer solutions. On the other hand, as the size of stock sheet becomes smaller, it becomes harder to find a placement of products with small unused area, since there are not enough small products to fill up the stock sheet.

5.7.5 Trade-off curves between m and solution quality

Finally, we conducted more detailed experiment to obtain the trade-off curve between m and the quality of the obtained solutions. We used two instances 40Vα and 40Vδ. The area of the stock sheet of 40Vδ is four times as large as that of 40Vα. Results are shown in Figures 5.1 and 5.2. In these figures, horizontal axis is m, and vertical axis shows the solution quality and CPU time in seconds. 40Vα-LP (resp., 40Vδ-LP) shows the average quality of obtained LP solutions (i.e., the numbers of applications can be fractional) for 40Vα (resp., 40Vδ), and 40Vα-IP (resp., 40Vδ-IP) shows the average quality of obtained

Table 5.3: Quality and time with various number of different patterns on our test instances

	$m = n$		$m = 0.8n$		$m = 0.6n$		$m = 0.4n$		
instance	quality	time	quality	time	quality	time	quality	time	f_{LB}†
20Sα	20.000	0.059	20.204	0.043	26.276	0.032	32.908	0.018	49
20Sβ	10.714	1.264	14.286	0.777	16.667	0.300	32.500	0.071	28
20Sγ	10.500	2.710	14.000	1.752	15.500	0.624	18.500	0.238	20
20Sδ	9.091	10.561	10.000	5.062	9.091	2.270	10.000	1.093	11
20Sε	22.000	10.650	28.000	21.647	34.000	11.542	36.000	7.213	5
20Lα	11.305	0.849	12.098	0.540	15.177	0.226	26.392	0.059	429
20Lβ	7.698	2.767	9.691	1.306	11.478	0.342	15.086	0.164	291
20Lγ	7.476	5.684	9.571	1.685	11.000	0.674	12.762	0.380	210
20Lδ	6.296	97.668	7.315	6.516	8.148	2.783	9.259	1.605	108
20Lε	6.226	97.668	7.547	26.189	7.736	19.458	8.302	9.656	53
20Vα	14.346	0.399	15.371	0.260	19.121	0.135	41.131	0.038	283
20Vβ	8.490	1.124	10.885	0.585	13.333	0.315	28.646	0.080	192
20Vγ	7.194	2.098	8.561	1.380	10.791	0.654	18.545	0.279	139
20Vδ	7.746	8.499	7.606	5.518	9.014	2.367	12.113	1.195	71
20Vε	8.286	34.356	8.571	21.362	9.143	14.572	10.000	7.225	35
30Sα	10.606	1.374	13.030	1.469	20.152	0.435	30.152	0.254	66
30Sβ	6.889	6.809	7.901	2.931	8.889	1.016	16.790	0.334	45
30Sγ	9.375	7.229	10.313	6.506	11.875	1.901	16.250	0.665	32
30Sδ	7.059	30.251	8.824	16.370	11.176	6.629	11.765	2.364	17
30Sε	25.000	106.728	25.000	67.912	25.000	32.616	25.000	16.230	8
30Lα	2.108	0.065	1.295	0.050	2.215	0.039	4.073	0.018	1244
30Lβ	5.783	13.063	6.821	4.320	10.178	0.864	15.488	0.236	563
30Lγ	5.774	21.359	6.929	4.787	9.036	1.336	12.948	0.440	407
30Lδ	5.000	61.168	5.625	19.060	6.971	4.460	8.558	1.535	208
30Lε	5.000	184.611	5.588	42.645	6.373	18.734	6.863	8.355	102
30Vα	14.150	2.501	18.518	1.105	21.942	0.372	36.279	0.082	479
30Vβ	12.369	5.547	13.569	4.519	17.108	0.822	28.400	0.276	325
30Vγ	8.468	11.880	9.447	5.892	12.894	1.501	16.894	0.654	235
30Vδ	6.250	56.867	7.833	23.923	10.000	5.217	12.250	2.618	120
30Vε	6.780	176.917	7.797	76.504	8.814	20.422	9.661	12.415	59

† f_{LB} is a lower bound of the number of required stock sheets computed by (5.2.8).

	$m = n$		$m = 0.8n$		$m = 0.6n$		$m = 0.4n$		
instance	quality	time	quality	time	quality	time	quality	time	f_{LB}†
40Sα	10.488	16.752	13.049	6.906	15.718	1.922	34.268	0.155	82
40Sβ	8.929	51.417	11.071	13.476	12.679	4.081	16.864	0.688	56
40Sγ	11.000	73.545	13.250	33.045	14.000	6.571	15.750	1.938	40
40Sδ	10.476	154.441	13.333	67.369	13.333	13.629	13.810	3.859	21
40Sε	22.000	225.421	22.000	101.199	23.000	46.489	22.000	19.962	10
40Lα	7.896	23.414	9.388	8.288	14.301	1.969	24.524	0.213	1030
40Lβ	5.908	51.554	8.054	17.153	10.615	3.280	13.763	0.727	699
40Lγ	5.307	120.834	7.069	24.252	8.950	4.655	12.416	1.312	505
40Lδ	5.233	422.831	6.357	76.808	7.209	11.777	8.682	3.680	258
40Lε	5.354	261.454	6.142	120.999	6.378	53.848	7.087	21.448	127
40Vα	8.880	20.867	11.639	4.437	16.475	1.469	31.330	0.283	366
40Vβ	7.218	38.728	8.669	10.465	11.514	2.793	17.115	0.646	248
40Vγ	7.598	77.630	8.994	28.141	11.620	4.462	15.419	1.331	179
40Vδ	6.413	185.486	7.500	34.984	8.370	17.397	10.109	25.059	92
40Vε	8.667	449.603	8.889	132.436	9.111	60.110	10.222	25.059	45
50Sα	10.000	216.917	13.465	29.109	14.752	6.544	23.338	0.778	101
50Sβ	9.565	308.406	11.739	47.813	12.609	8.750	16.232	2.039	69
50Sγ	10.000	410.122	12.400	85.796	13.600	17.967	14.400	4.206	50
50Sδ	12.308	564.631	13.077	157.077	13.077	36.010	13.462	15.529	26
50Sε	14.615	784.706	16.154	280.447	16.154	108.142	15.385	49.327	13
50Lα	7.572	168.741	9.863	37.406	13.336	4.682	21.708	0.823	1211
50Lβ	6.253	213.977	8.382	37.713	10.061	10.830	14.002	2.499	822
50Lγ	6.061	519.849	7.290	46.188	8.603	18.621	10.724	4.105	594
50Lδ	5.677	1120.034	6.403	147.052	7.393	48.016	8.515	12.327	303
50Lε	5.772	1856.302	6.376	520.601	6.846	130.716	6.913	37.791	149
50Vα	8.843	111.389	11.316	30.402	16.413	4.289	32.327	0.451	828
50Vβ	6.762	165.888	9.253	45.889	12.100	7.265	18.345	1.596	562
50Vγ	6.330	441.155	8.103	100.119	10.493	16.128	14.335	3.973	406
50Vδ	6.425	1020.919	7.923	169.479	8.986	39.370	11.014	7.945	207
50Vε	6.471	1649.879	7.353	411.934	8.137	102.056	9.020	38.361	102

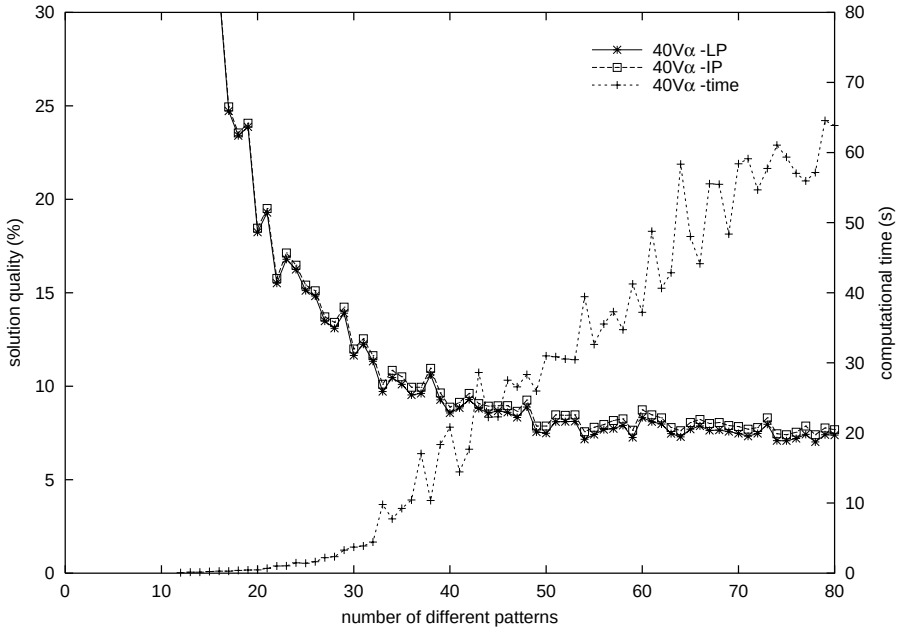


Figure 5.1: Trade-off between m and solution quality for 40Vα

IP solutions (i.e., the numbers of applications must be integer) for 40Vα (resp., 40Vδ). 40Vα-time and 40Vδ-time show the average CPU times in seconds for ten trials. When m is very small (i.e., $m \leq 11$ for 40Vα and $m \leq 2$ for 40Vδ), we could not find initial feasible solutions. From Figures 5.1 and 5.2, we observe that the computational time tends to increase and the solution quality improves as m increases. For larger m, the improvement in quality becomes tiny while the computational time is increasing steadily. Note that, if the numbers of applications can be fractional, it is known that an optimal solution for 2DCSP uses at most n different patterns. Nevertheless, our obtained LP solutions for these instances with $m = 40$ are slightly worse than those with larger m. From these observations, there is still room for improvement in our neighborhood search. We also observe that the gap between LP and IP solutions for 40Vδ are more significant than the gap for 40Vα.

5.8 Conclusion

In this chapter, we considered the two-dimensional cutting stock problem using a given number of different patterns m. As this is an intractable combinatorial optimization problem, we proposed a local search algorithm, which is based on linear programming techniques. In this algorithm, we utilized heuristic algorithms to solve three subproblems; i.e., the problem to compute the numbers of applications for Π, the two-dimensional bin

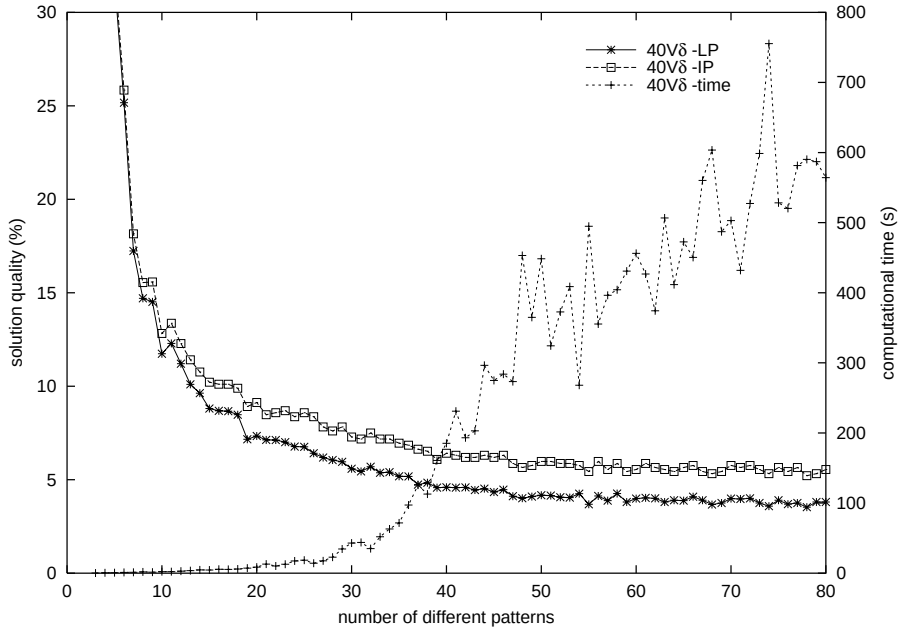


Figure 5.2: Trade-off between m and solution quality for 40Vδ

packing problem and the rectangle packing problem.

To see the performance of our local search algorithm, we conducted some computational experiments with randomly generated test instances of 2DCSP. We first confirmed the effectiveness of our enhanced neighborhood, which utilized basic, redundancy reduction, filling-up and replacement operations. We also computed the trade-off curves between m and the quality of the obtained solution.

As a future work, we are planning to improve the solution quality by introducing more efficient neighborhood search and by incorporating more sophisticated metaheuristic algorithms.

Chapter 6

Conclusion

Throughout this thesis, we have considered the developments of practical algorithms for some standard problems that can include a wide range of specific variants of cutting and packing problems. The studies in this thesis are summarized as follows.

First, in Chapters 1 and 2, we reviewed variants of cutting and packing problems and explained basic techniques and algorithms for these problems. Cutting and packing problems appear in many real-world applications, and there are numerous variants which should be solved. However, it would be impossible to develop individual algorithms for all of such problems. We hence proposed a standard problem of the rectangle packing problem called the rectangle packing problem with general spatial costs (RPGSC). Introducing various cost functions and defining them appropriately, many variants of rectangle packing problem and scheduling problems can be formulated in this form.

In Chapter 3, we developed practical approximate algorithms for RPGSC. We adopted the sequence pair representation [90] as the coding scheme in our algorithm, and proposed two different decoding algorithms based on dynamic programming and encoding algorithms based on the plane sweep technique. The first decoding algorithm CMPF was a generalization of the existing decoding algorithms proposed in [105, 106] in that it could deal with general spatial costs. Another decoding algorithm CBP relaxed the constraints of a sequence pair slightly and found a packing not worse than that obtained by CMPF. In order to find a good coded solution, we utilized metaheuristic algorithms based on local search. For an efficient local search, we made use of the critical paths, which denote the bottleneck of the current packing, to reduce the size of neighborhoods. From computational experiments for various specific types of the rectangle packing problem and a real-world scheduling problem, it turned out that our algorithm can handle a wide range of problems. Furthermore, our algorithm was competitive with other existing heuristic algorithms specially tailored to each specific problem.

Next, in Chapter 4, we tried an enhancement of our local search algorithm for RPGSC. In our local search algorithm for RPGSC, we must generate numerous coded solutions (sequence pairs) and evaluate all of them. In Chapter 4, we proposed new decoding algorithms, called ESM and ELSM, to evaluate all coded solutions in various neighborhoods efficiently. In general, solutions in a neighborhood have similar structure and we made use of it in these algorithms. The amortized computational time of a new decoding algorithm ESM (resp., ELSM) per one coded solution was $O(1)$ (resp., $O(\log n)$), if applied to some specific problems including area minimization, strip packing, two-dimensional knapsack and so on. The usefulness of our decoding algorithms was then demonstrated through computational experiments for several variants of the rectangle packing problem and scheduling problem.

In Chapter 5, we focused on the two-dimensional cutting stock problem (2DCSP). 2DCSP is an important problem for numerous real-world applications such as the manufacturing industry, and many algorithms were proposed for this problem. We first proposed a new problem called the two-dimensional cutting stock problem with a given number of different patterns (2DCSPm) and then developed a local search algorithm for this problem. In this algorithm, we utilized heuristic algorithms to solve three subproblems; i.e., the problem to compute the numbers of applications for the set of cutting patterns Π, the two-dimensional bin packing problem and the rectangle packing problem. To see the performance of our local search algorithm, we conducted computational experiments with randomly generated test instances of 2DCSP. We first confirmed the effectiveness of our enhanced neighborhood, which utilized basic, redundancy reduction, filling-up and replacement operations. We also computed the trade-off curves between m and the quality of the obtained solution.

In recent years, as systems in real applications have become more sophisticated, problems have become more complicated than those simple local search and metaheuristic algorithms can handle. In order to cope with these phenomena, many hybrid algorithms have been studied; i.e., various heuristic algorithms and exact algorithms are introduced to local search and metaheuristic algorithms, or basic principle of metaheuristic algorithms are combined together to realize more powerful tools. However, we must take into account the fact that these hybridization of algorithms may often spoil the flexibility and simplicity of local search and metaheuristic algorithms. In the real-world applications, new problems are continuously arising, and it would be impossible to develop individual algorithms for all of such problems. Therefore, we believe it is very important to develop practical algorithms for appropriately chosen standard problems, each of which can handle a wide range of specific problems. The author hopes that the study in this thesis will be helpful for the developments of such useful algorithms.

Bibliography

[1] E.H.L. Aarts and J.K. Lenstra (eds.), *Local Search in Combinatorial Optimization,* John Wiley and Sons (1997).

[2] N.R. Achuthan, L. Caccetta and S.P. Hill, "An improved branch-and-cut algorithm for the capacitated vehicle routing problem," *Transportation Science* 37 (2003) 153–169.

[3] M. Adamowicz and A. Albano, "Nesting two-dimensional shapes in rectangular modules," *Computer Aided Design* 8 (1976) 27–33.

[4] P.K. Agarwal, M. Sharir and P. Shor, "Sharp upper and lower bounds on the length of general Davenport-Schinzel sequences," *Journal of Combinatorial Theory Series A* 52 (1989) 228–274.

[5] R.K. Ahuja, D.S. Hochbaum and J.B. Orlin, "Solving the convex cost integer dual network flow problem," *Management Science* 49 (2003) 950–964.

[6] R.K. Ahuja, D.S. Hochbaum and J.B. Orlin, "A cut-based algorithm for the nonlinear dual of the minimum cost network flow problem," *Algorithmica* (to appear).

[7] R.K. Ahuja, J.B. Orlin and D. Sharma, "Very large-scale neighborhood search," *International Transactions in Operational Research* 7 (2000) 301–317.

[8] B.S. Baker, E.G. Coffman Jr. and R.L. Rivest, "Orthogonal packing in two dimensions," *SIAM Journal on Computing* 9 (1980) 846–855.

[9] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry,* Springer-Verlag (1997).

[10] J.O. Berkey and P.Y. Wang, "Two dimensional finite bin packing algorithms," *Journal of the Operational Research Society* 38 (1987) 423–429.

[11] E.E. Bischoff and M.S.W. Ratcliff, "Loading multiple pallets," *Journal of the Operational Research Society* 46 (1995) 1322–1336.

[12] J. Blażewicz and R. Walkowiak, "A local search approach for two-dimensional irregular cutting," *OR Spectrum* 17 (1995) 93–98.

[13] R.M. Brady, "Optimization strategies gleaned from biological evolution," *Nature* 317 (1985) 804–806.

[14] P. Brucker, *Scheduling Algorithms,* Springer-Verlag (1995).

[15] E.K. Burke and S. Petrovic, "Recent research directions in automated timetabling," *European Journal of Operational Research* 140 (2002) 266–280.

[16] A. Caprara and P. Toth, "Lower bounds and algorithms for the 2-dimensional vector packing problem," *Discrete Applied Mathematics* 111 (2001) 231–262.

[17] L. Cavique, C. Rego and I. Themido, "Case oriented paper – Subgraph ejection chains and tabu search for the crew scheduling problem," *Journal of the Operational Research* 50 (1999) 608–616.

[18] F. Chauny and R. Loulou, "LP-based method for the multi-sheet cutting stock problem," *INFOR*, 32 (1994) 253–264.

[19] B. Chazelle, "The bottom-left bin-packing heuristic: an efficient implementation," *IEEE Transactions on Computers* 32 (1983) 697–707.

[20] F.K.R. Chung, M.R. Garey and D.S. Johnson, "On packing two-dimensional bins," *SIAM Journal on Algebraic and Discrete Methods* 3 (1982) 66–76.

[21] E.G. Coffman Jr., M.R. Garey D.S. Johnson and R.E.Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM Journal on Computing* 9 (1980) 801–826.

[22] H. Cohn and M. Fielding, "Simulated annealing: searching for an optimal temperature schedule," *SIAM Journal on Optimization* 8 (1999) 779–802.

[23] C.R. Collins and K. Stephenson, "A circle packing algorithm," *Computational Geometry* 25 (2003) 233–256.

[24] V.D. Cung, M. Hifi and B.L. Cun, "Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm," *International Transactions in Operational Research* 7 (2000) 185–210.

[25] L. Davis, *Handbook of Genetic Algorithms,* Van Nostrand Reinhold (1991).

[26] J.A. Diaz and E. Fernandez, "A branch-and-price algorithm for the single source capacitated plant location problem," *Journal of the Operational Research Society* 53 (2002) 728–740.

[27] E.W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik* 1 (1959) 269–271.

[28] K.A. Dowsland and W.B. Dowsland, "Packing problems," *European Journal of Operational Research* 56 (1992) 2–14.

[29] K.A. Dowsland, S. Vaid and W.B. Dowsland, "An algorithm for polygon placement using a bottom-left strategy," *European Journal of Operational Research* 141 (2002) 371–381.

[30] G. Dueck and T. Scheuer, "Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing," *Journal of Computational Physics* 90 (1990) 161–175.

[31] H. Dyckhoff, "A typology of cutting and packing problems," *European Journal of Operational Research* 44 (1990) 145–159.

[32] H. Dyckhoff and U. Finke, *Cutting and Packing in Production and Distribution,* Physica Verlag, Heidelberg (1992).

[33] U. Faigle and W. Kern, "Some convergence results for probabilistic tabu search," *ORSA Journal on Computing* 4 (1992) 32–37.

[34] A.A. Farley, "Practical adaptations of the Gilmore-Gomory approach to cutting stock problems," *OR Spectrum* 10 (1998) 113–123.

[35] O. Faroe, D. Pisinger and M. Zachariasen, "Guided local search for final placement in VLSI design," *Journal of Heuristics* 9 (2003) 269–295.

[36] T.A. Feo and M.G.C. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Operations Research Letters* 8 (1989) 67–71.

[37] H. Foerster and G. Wäscher, "Pattern reduction in one-dimensional cutting stock problems," *International Journal of Production Research* 38 (2000) 1657–1676.

[38] M.R. Garey, R.L. Graham, D.S. Johnson and A.C. Yao, "Resource constrained scheduling as generalized bin packing," *Journal of Combinatorial Theory Series A* 21 (1976) 257–298.

[39] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman (1979).

[40] S. Geman and D. Geman, "Stochastic relaxation, Gibbs distribution, and the Bayesian restoration of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6 (1984) 721–741.

[41] S.C. Ghosh, B.P. Sinha and N. Das, "Channel assignment using genetic algorithm based on geometric symmetry," *IEEE Transactions on Vehicular Technology* 52 (2003) 860–875.

[42] P.C. Gilmore and R.E. Gomory, "A linear programming approach to the cutting-stock problem," *Operations Research* 9 (1961) 849–859.

[43] P.C. Gilmore and R.E. Gomory, "A linear programming approach to the cutting-stock problem – Part II," *Operations Research* 11 (1963) 863–888.

[44] P.C. Gilmore and R.E. Gomory, "Multistage cutting stock problems of two and more dimensions," *Operations Research* 13 (1965) 94–119.

[45] C.A. Glass, C.N. Potts and V.A. Strusevich, "Scheduling batches with sequential jobs processing for two-machine flow and open shops," *INFORMS Journal on Computing* 13 (2001) 120–137.

[46] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers and Operations Research* 13 (1986) 533–549.

[47] F. Glover, "Genetic algorithms and scatter search: unsuspected potentials," *Statistics and Computing* 4 (1994) 131–140.

[48] F. Glover, "Ejection chains reference structures and alternating path methods for traveling salesman problems," *Discrete Applied Mathematics* 65 (1996) 223–253.

[49] F. Glover and S. Hanafi, "Tabu search and finite convergence," *Discrete Applied Marhematics* 119 (2002) 3–36.

[50] F. Glover and M. Laguna, *Tabu Search,* Kluwer Academic Publishers (1997).

[51] A.M. Gomes and J.F. Oliveira, "A 2-exchange heuristic for nesting problems," *European Journal of Operational Research* 141 (2002) 359–370.

[52] J. Gu and X. Huang, "Efficient local search with search space smoothing: a case study of the traveling salesman problem (TSP)," *IEEE Transactions on Systems, Man, and Cybernetics* 24 (1994) 728–735.

[53] P.N. Guo, T. Takahashi, C.K. Cheng and T. Yoshimura, "Floorplanning using a tree representation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 20 (2001) 281–289.

[54] R.E. Haessler, "Controlling cutting pattern changes in one-dimensional trim problems," *Operations Research* 23 (1975) 483–493.

[55] S. Hanafi, "On the convergence of tabu search," *Journal of Heuristics* 7 (2001) 47–58.

[56] S. Hartmann, "Packing problem and project scheduling models: an integrating perspective," *Journal of Operational Research Society* 51 (2000) 1083–1092.

[57] J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence,* The University of Michigan Press (1975), and MIT Press (1992).

[58] E. Hopper and B.C.H. Turton, "An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem," *European Journal of Operational Research* 128 (2001) 34–57.

[59] E. Hopper and B.C.H. Turton, "A review on the application of meta-heuristic algorithms to 2D strip packing problems," *Artificial Intelligence Review* 16 (2001) 257–300.

[60] T. Ibaraki, *Enumerative Approaches to Combinatorial Optimization,* Annals of Operations Research, Vols. 10 and 11, J.C. Baltzer A.G. (1987).

[61] T. Ibaraki, S. Imahori, M. Kubo, T. Masuda, T. Uno and M. Yagiura, "Effective local search algorithms for routing and scheduling problems with general time window constraints," *Transportation Science* (to appear).

[62] O.H. Ibarra and C.E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *Journal of the ACM* 22 (1975) 463–468.

[63] S. Imahori, M. Yagiura and T. Ibaraki, "Local search algorithms for the rectangle packing problem with general spatial costs," *Mathematical Programming* 97 (2003) 543–569.

[64] S. Imahori, M. Yagiura and T. Ibaraki, "Improved local search algorithms for the rectangle packing problem with general spatial costs," submitted for publication (available at http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing).

[65] S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two dimensional cutting stock problem," *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics* Volume IX (2003) 334–339.

[66] S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two-dimensional cutting stock problem with a given number of different patterns," submitted for publication (available at http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing).

[67] B. Jaumard, O. Marcotte, C. Meyer and T. Vovor, "Comparison of column generation models for channel assignment in cellular networks," *Discrete Applied Mathematics* 112 (2001) 217–240.

[68] D.S. Johnson, "Local optimization and the traveling salesman problem," in: M.S. Peterson, ed., *Automata, Languages and Programming*, vol. 443 of *Lecture Notes in Computer Science*, Springer-Verlag (1990) 446–461.

[69] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey and R.L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing* 3 (1974) 299–325.

[70] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal* 49 (1970) 291–307.

[71] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, "Optimization by simulated annealing," *Science* 220 (1983) 671–680.

[72] D. Klabjan, E.L. Johnson, G.L. Nemhauser, E. Gelman and S. Ramaswamy, "Airline crew scheduling with time windows and plane count constraints," *Transportation Science* 36 (2002) 337–348.

[73] D.E. Knuth, *The Art of Computer Programming: Vol 3 Sorting and searching,* Addison Wesley (1998).

[74] L.G. Kroon and L.W.P. Peeters, "A variable trip time model for cyclic railway timetabling," *Transportation Science* 37 (2003) 198–212.

[75] J.B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society* 7 (1956) 48–50.

[76] A.N. Letchford and A. Amaral, "Analysis of upper bounds for the pallet loading problem," *European Journal of Operational Research* 132 (2001) 582–593.

[77] D. Liu and H. Teng, "An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles," *European Journal of Operational Research* 112 (1999) 413–420.

[78] A. Lodi, S. Martello and M. Monaci, "Two-dimensional packing problems: A survey," *European Journal of Operational Research* 141 (2002) 241–252.

[79] A. Lodi, S. Martello and D. Vigo, "Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems," *INFORMS Journal on Computing* 11 (1999) 345–357.

[80] A. Lodi, S. Martello and D. Vigo, "Recent advances on two-dimensional bin packing problems," *Discrete Applied Mathematics* 123 (2002) 379–396.

[81] A. Lodi and M. Monaci, "Integer linear programming models for 2-staged two-dimensional knapsack problems," *Mathematical Programming* 94 (2003) 257–278.

[82] M. Lundy and A. Mees, "Convergence of an annealing algorithm," *Mathematical Programming* 34 (1986) 111–124.

[83] O. Marcotte, "The cutting stock problem and integer rounding," *Mathematical Programming* 33 (1985) 82–92.

[84] S. Martello and D. Vigo, "Exact solution of the two-dimensional finite bin packing problem," *Management Science* 44 (1998) 388–399.

[85] S. Martello, M. Monaci and D. Vigo, "An exact approach to the strip-packing problem," *INFORMS Journal on Computing* 15 (2003) 310–319.

[86] A.J. Miller, G.L. Nemhauser and M.W.P. Savalsbergh, "On the polyhedral structure of a multi-item production planning model with setup time," *Mathematical Programming* 94 (2002) 375–406.

[87] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers and Operations Research* 24 (1997) 1097–1100.

[88] R. Morabito and S. Morales, "A simple and effective recursive procedure for the manufacturer's pallet loading problem," *Journal of the Operational Research Society* 49 (1998) 819–828.

[89] M. Mukherjee and K. Chakraborty, "A polynomial-time optimization algorithm for a rectilinear partitioning problem with applications in VLSI design automation," *Information Processing Letters* 83 (2002) 41–48.

[90] H. Murata, K. Fujiyoshi, S. Nakatake and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Transactions on Computer Aided Design* 15-12 (1996) 1518–1524.

[91] A. Nagao, T. Sawa, Y. Shigehiro, I. Shirakawa and T. Kambe, "A new approach to rectangle-packing," *Electronics and Communications in Japan* 83 (2000) 94–104.

[92] S. Nakatake, K. Fujiyoshi, H. Murata and Y. Kajitani, "Module placement on BSG-structure and IC layout applications," *Proceedings of International Conference on Computer Aided Design* (1996) 484–491.

[93] K. Nonobe and T. Ibaraki, "Formulation and tabu search algorithm for the resource constrained project scheduling problem," in: *Essays and Surveys in Metaheuristics,* Kluwer Academic Publishers (2001) 557–588.

[94] H. Okano, "A scanline-based algorithm for the 2D free-form bin packing problem," *Journal of the Operations Research Society of Japan* 45 (2002) 145–161.

[95] I.H. Osman and J.P. Kelly (eds.), *Meta-Heuristics: Theory and Applications,* Kluwer Academic Publishers (1996).

[96] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and complexity,* Prentice-Hall (1982).

[97] D. Pisinger, "Heuristics for the container loading problem," *European Journal of Operational Research* 141 (2002) 382–392.

[98] R.C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal* 36 (1957) 1389–1401.

[99] S. Rajagopalan and J.M. Swaminathan, "A coordinated production planning model with capacity expansion and inventory management," *Management Science* 47 (2001) 1562–1580.

[100] J. Riehme, G. Scheithauer and J. Terno, "The solution of two-stage guillotine cutting stock problems having extremely varying order demands," *European Journal of Operational Research* 91 (1996) 543–552.

[101] G. Rudolph, "Convergence analysis of canonical genetic algorithms," *IEEE Transactions on Neural Networks* 5 (1994) 96–101.

[102] K. Sakanushi, Y. Kajitani and D.P. Mehta, "The quarter-state-sequence floorplan representation," *IEEE Transactions on Circuits and Systems* 50 (2003) 376–386.

[103] B. Selman, H.A. Kautz and B. Cohen, "Noise strategies for improving local search," *Proceedings of 12th National Conference on Artificial Intelligence* (1994) 337–343.

[104] P.E. Sweeney and E.R. Paternoster, "Cutting and packing problems: A categorized, application-orientated research bibliography," *Journal of the Operational Research Society* 43 (1992) 691–706.

[105] 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.

[106] X. Tang, R. Tian and D.F. Wong, "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 (2001) 1406–1413.

[107] X. Tang and D.F. Wong, "Fast-SP: a fast algorithm for block placement based on sequence pair," *Proceedings of Asia and South Pacific Design Automation Conference* (2001) 521–526.

[108] S. Umetani, M. Yagiura and T. Ibaraki, "An LP-based local search to the one dimensional cutting stock problem using a given number of cutting patterns," *IEICE Transactions on Fundamentals* E86-A (2003) 1093–1102.

[109] R.A. Valdés, A. Parajón, and J.M. Tamarit, "A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems," *Computers and Operations Research* 29 (2002) 925–947.

[110] R.A. Valdés, A. Parajón, and J.M. Tamarit, "A computational study of LP-based heuristic algorithms for two-dimensional guillotine cutting stock problems," *OR Spectrum* 24 (2002) 179–192.

[111] F. Vanderbeck, "Computational study of a column generation algorithm for bin packing and cutting stock problems," *Mathematical Programming* 86 (1999) 565–594.

[112] F. Vanderbeck, "A nested decomposition approach to a three-stage, two-dimensional cutting-stock problem," *Management Science* 47 (2001) 864–879.

[113] V. Verter and A. Dasci, "The plant location and flexible technology acquisition problem," *European Journal of Operational Research* 136 (2002) 366–382.

[114] C. Voudoris and E. Tsang, "Guided local search and its application to the traveling salesman problem," *European Journal of Operational Research* 113 (1999) 469–499.

[115] H. Wang, W. Huang, Q. Zhang and D. Xu, "An improved algorithm for the packing of unequal circles within a larger containing circle," *European Journal of Operational Research* 141 (2002) 440–453.

[116] Y.L. Wu, W. Huang, S. Lau, C.K. Wong and G.H. Young, "An effective quasi-human based heuristic for solving the rectangle packing problem," *European Journal of Operational Research* 141 (2002) 341–358.

[117] M. Yagiura and T. Ibaraki, "On metaheuristic algorithms for combinatorial optimization problems," *Systems and Computers in Japan* 32 (2001) 33–55.

[118] M. Yagiura, T. Ibaraki and F. Glover, "An ejection chain approach for the generalized assignment problem," *INFORMS Journal on Computing* (to appear).

[119] S. Zionts, "The criss-cross method for solving linear programming problems," *Management Science* 15 (1969) 426–445.

A List of Author's Work

Journals

1. S. Imahori, M. Yagiura and T. Ibaraki, "Local search algorithms for the rectangle packing problem with general spatial costs," *Mathematical Programming* 97 (2003) 543–569.

2. T. Ibaraki, S. Imahori, M. Kubo, T. Masuda, T. Uno and M. Yagiura, "Effective local search algorithms for routing and scheduling problems with general time window constraints," *Transportation Science* (to appear).

International Conferences with Review

1. S. Imahori, M. Yagiura and T. Ibaraki, "Local search heuristics for the rectangle packing problem with general spatial costs," *Proceedings of the 4th Metaheuristics International Conference* (2001) 471–476.

2. S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two dimensional cutting stock problem," *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics* Volume IX (2003) 334–339.

3. T. Ibaraki, S. Imahori, K. Nonobe, K. Sobue, T. Uno and M. Yagiura, "An iterated local search algorithm for routing and scheduling problem with convex time penalty functions," *Proceedings of the 5th Metaheuristics International Conference* (2003) 33.

4. S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two dimensional cutting stock problem with a given number of patterns," *Proceedings of the 5th Metaheuristics International Conference* (2003) 35.

Unpublished Manuscripts

1. S. Imahori, M. Yagiura and T. Ibaraki, "Improved local search algorithms for the rectangle packing problem with general spatial costs," submitted for publication (available at http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing).

2. S. Imahori, M. Yagiura, S. Umetani, S. Adachi and T. Ibaraki, "Local search algorithms for the two-dimensional cutting stock problem with a given number of different patterns," submitted for publication (available at http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing).