Construction Heuristics for the Rectilinear Block Packing Problem

Yannan HU

Construction Heuristics for the Rectilinear Block Packing Problem

Yannan HU

Department of Computer Science and Mathematical Informatics
Graduate School of Information Science
Nagoya University
Nagoya 464-8601, Japan

February, 2016

Doctoral Dissertation

submitted to Graduate School of Information Science, Nagoya University

in partial fulfillment of the requirement for the degree of

DOCTOR OF INFORMATION SCIENCE

Preface

Combinatorial optimization is one of the most active areas in operations research, computer science and discrete mathematics. It appears in various fields such as operations research, applied mathematics and theoretical computer science. Combinatorial optimization problems involve various applications, such as airline crew scheduling, VLSI design and network design. A variety of real-life problems can be abstracted as combinatorial optimization problems. Some of the classical problems in combinatorial optimization such as minimum spanning trees, shortest paths, network flows, matchings, etc., have polynomial time algorithms. However, there are numerous combinatorial optimization problems such as the knapsack problem, traveling salesman problem, scheduling problem, packing problem, which are proved to be NP-hard. Under the widely believed conjecture that $P \neq NP$, finding their exact solutions is prohibitively time consuming in the worst case. This negative aspect is not caused by missing more clever approaches to the problems but is indeed a general property of the NP-hard problems. Consequently, we have to look for other ways to deal with this situation. In particular, in many practical applications, it is not really necessary to find exact optimal solutions, and sufficiently good solutions would be enough. Hence, the approximation and heuristic algorithms that compute sufficiently good solutions in reasonable time become very important.

There are several techniques for designing approximation algorithms and heuristics such as greedy algorithms and local search algorithms. These ideas are useful when devising approximation or heuristic algorithms for NP-hard problems. Algorithms derived from these techniques are often easy to implement and run in short times in practice. Recently, there have been an enormous number of studies on various types of heuristics and metaheuristics such as simulated annealing, genetic algorithms, tabu search, evolutionary computation and greedy randomized adaptive search procedures (GRASP). When more quality is needed and more computation time is available, these techniques are often very effective and yield good results in practice.

In this thesis, we focus on the two-dimensional packing problem. It is among the classical packing problems and a series of approaches have been developed. There are

three major types of two-dimensional packing problems: the bin packing problem, the strip packing problem and the area minimization problem. In a two-dimensional packing problem, we are usually given items (rectangles, circles, or arbitrarily shaped polygons) and containers. We focus on the rectilinear block strip packing problem, where a rectilinear block is a polygonal block whose interior angle is either 90° or 270°. Given a set of rectilinear blocks and a rectangular container with fixed width and infinite height (called a *strip*), the task is to pack all the rectilinear blocks into the container so as to minimize the height of the container.

Rectilinear blocks have similar properties with rectangles since they can be represented by sets of rectangles whose relative positions are fixed. However, rectilinear blocks are more flexible. Any arbitrarily shaped polygon can be approximately represented by a rectilinear block. Consequently, the methods developed for the rectilinear block packing problem may also work for a large variety of other variants of the strip packing problem.

There are some papers for the rectilinear block packing problem in the literature. Most of them only deal with small-scale instances. However, for industrial application purposes, good solutions for large-scale instances are often necessary. We observe that construction heuristics tend to perform well on large-scale instances and it is possible to design efficient implementations to make their running time very short. Hence, in this research, we develop construction heuristics for the rectilinear block packing problem.

We first generalize two representative construction heuristics for the rectangle packing problem to solve the rectilinear block packing problem and propose efficient implementations for them. Algorithms with different implementations devised to solve the same problem often differ dramatically in their efficiency. The design and analysis of efficient data structures have long been recognized as important subjects in implementation. We then design more efficient implementations utilizing sophisticated data structures to keep the necessary information dynamically so that the running time is significantly reduced. We perform analysis of the running time of our algorithms based on different implementations from both sides of theory and practice. We also analyze the performance of the developed heuristic algorithms and propose a new construction heuristic as a bridge between the two algorithms.

We hope that our algorithms are useful in practical applications and the developed implementations and techniques can provide insight to help improve the design of algorithms for solving combinatorial optimization problems.

February, 2016
Yannan HU

Acknowledgments

This thesis has benefited from discussions and suggestions of many people, whom I would like to acknowledge here.

I would like to start my acknowledgements by thanking Professor Mutsunori Yagiura of Nagoya 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.

Several enthusiastic discussions with Professor Shinji Imahori of Chuo University were quite exciting and invaluable experience for me. His heartful and earnest guidance has encouraged me all the time.

I would like to express my sincere appreciation to Associate Professor Hideki Hashimoto of Tokyo University of Marine Science and Technology, who guided me in research and gave me numbers of valuable comments. The comments by Professor Takeaki Uno of National Institute of Informatics are also greatly appreciated.

I am heartily grateful to Professor Tomio Hirata of Nagoya University for his generous help and support, which makes this paper complete smoothly.

I am also thankful to Professor Takafumi Kanamori, Yo Matsubara and Masahiko Sakai of Nagoya University for serving on my dissertation committee. Professor Masahiko Sakai gave me lots of precious comments that significantly improved the quality of this manuscript.

To my personal friend Marco Milano of University of Wuppertal, I owe special thanks for helpful suggestions and valuable technical support, in particular in the preparation of figures. I wish to express my gratitude to all members in Yagiura and Hirata laboratories of Nagoya University for many enlightening discussions on the area of this work and their warm friendship in the period I studied in Japan.

Last, 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

List of Algorithms

Chapter 1

Introduction

Combinatorial optimization [63] is one of the most active areas in operations research, theoretical computer science and discrete mathematics. Combinatorial optimization problems involve various applications, such as airline crew scheduling, VLSI design and network design. Many real-life problems can be abstracted mathematically as minimization or maximization problems.

Some of the classical problems in combinatorial optimization such as minimum spanning trees, shortest paths, network flows, matchings, etc., have polynomial time algorithms. However, there are numerous combinatorial optimization problems such as the knapsack problem, traveling salesman problem, scheduling problem, packing problem, which are proved to be NP-hard. Under the widely believed conjecture that P $\neq$ NP, finding their exact solutions are prohibitively time consuming in the worst case. Consequently, we have to look for other ways to deal with this situation. In particular, in many practical applications, it is not really necessary to find exact optimal solutions, and sufficiently good solutions would be satisfactory. Hence, the approximation or heuristic algorithms that compute sufficiently good solutions in reasonable time become very important.

Packing problems are classic combinatorial optimization problems and known to be NP-hard. Many packing problems are related to real-world applications and they have been studied for a long time from both theoretical and practical points of view. In this thesis, we focus on the *rectilinear block strip packing problem*, where a rectilinear block is a polygonal block whose interior angle is either 90° or 270°. Given a set of rectilinear blocks, each with a deterministic shape and size, and a rectangular container (also called a strip) with fixed width and unrestricted height, the task is to pack all the items orthogonally without overlap into the container so as to minimize height of the container. Figure 1.1 shows an example of the rectilinear block packing problem. The layout on the right is an example packing layout after packing all seven rectilinear blocks into the container given

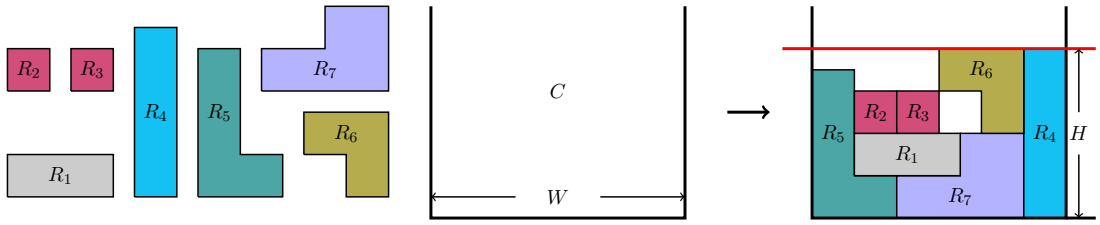


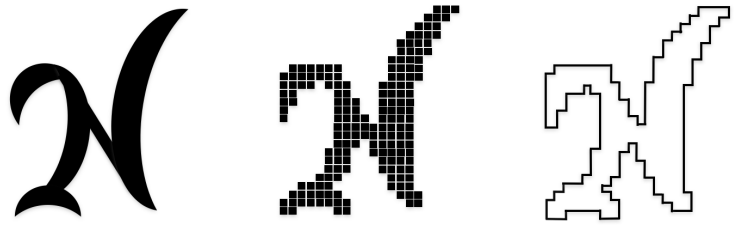
Figure 1.1: An instance of the rectilinear block packing problem and a solution

on the left of the figure. The objective is to minimize the height of the container.

According to the improved typology of Wäscher et al. [92], strip packing problems are categorized into the two dimensional open dimension problem (2D ODP) with a single variable dimension. The rectilinear block strip packing problem is among classical packing problems and is known to be NP-hard [9]. It involves many industrial applications, such as VLSI design, timber/glass cutting, and newspaper layout.

Rectilinear blocks have similar properties to rectangles since they can be represented by sets of rectangles whose relative positions are fixed. However, rectilinear blocks are more flexible. A rectangle can be treated as a special case of a rectilinear block and any arbitrarily shaped polygon can be approximately represented by a rectilinear block. In computer graphics, a bitmap image is a dot matrix data structure representing a grid of pixels. An arbitrarily shaped polygon, no matter how complex it is in shape, can be approximately represented as a bitmap image. The resulting layout of a bitmap image can be treated as a rectilinear block by considering each dot of a bitmap image as a square. Figure 1.2 shows an example of representing an arbitrarily shaped polygon by a rectilinear block. Consequently, methods developed for the rectilinear block packing problem will also work for a large variety of other variants of the strip packing problem. We restrict the target of this thesis to the rectilinear block packing problem.

It is difficult to represent the relationships among rectilinear blocks than those among rectangles. Several structures have been designed to represent the relationships among rectilinear blocks such as the *bounded-sliceline grid* (*BSG*) [78, 87], *sequence-pairs* [32, 33, 56, 94], O-tree [82], B*-tree [93], *transitive closure graph* (*TCG*) [67] and *corner block list* (*CBL*) [69]. Several heuristics have been proposed for the rectilinear block packing problem based on these structures. Most of those methods perform well for small-scale instances with up to 100 items in reasonable running time, and they are especially effective for instances consisting of rectilinear blocks that have simple shapes (e.g., L- or T-shape). However, in many industrial applications such as VLSI design, good solutions are often



arbitrarily shaped polygon (a) bitmap image (b) rectilinear block (c)

Figure 1.2: An arbitrarily shaped object represented as a bitmap shape

necessary for large-scale instances or instances with rectilinear blocks having complicated shapes. In this thesis, we focus on dealing with large-scale instances of the rectilinear block packing problem.

Many efficient algorithms have been proposed for the rectangle strip packing problem. The *bottom-left algorithm* [9] and the *best-fit algorithm* [19] are known as the most remarkable works among existing construction heuristics for the rectangle packing problem. We study these two construction heuristics and observe that construction algorithms tend to obtain better performance when the sizes of instances increase, and it is possible to design efficient implementations for construction algorithms to reduce their running time. Inspired by these construction algorithms, we develop construction heuristics for the rectilinear block packing problem.

We adopt the *bottom-left strategy* as the main strategy of our construction algorithms. In this strategy, starting from an empty layout, items are packed into the container one by one, and whenever a new item is packed into the container, it is placed at the *bottom-left* (*BL*) *position* relative to the current layout. The BL position of a new item relative to a packing layout is defined as the leftmost location among the lowest *bottom-left stable feasible positions*, where a bottom-left stable feasible position is a location where the new item can be placed without overlap and cannot be moved leftward or downward. The technique of *no-fit polygon* [7] is very useful when determining whether two polygons overlap each other in two-dimensional space. We take the concept of no-fit polygons as a crucial technique for packing rectilinear blocks into container without overlap and design methods of searching for the BL position of a rectilinear block.

We first generalize the bottom-left and the best-fit algorithms for the rectangle packing problem to solve the rectilinear block packing problem and propose efficient implementations for these two generalized algorithms, using an efficient method for finding BL positions that was originally proposed for rectangle packing. Different implementations

devised to solve the same problem often differ dramatically in their efficiency. The design and analysis of efficient data structures have long been recognized as important subjects in implementation. We consider some sophisticated data structures, specially tailored for our problem, and design more efficient implementations of the bottom-left and the best-fit algorithms. We use sophisticated data structures that keep the information dynamically so that the BL position of each item can be found in sub-linear amortized time. We also analyze time complexities of these algorithms with the new implementations.

A series of experiments on a set of instances that are generated from nine benchmark instances are performed to analyze the performance of our algorithms from both sides of occupation rate and running time. Our algorithms often perform well for large instances. We compare the running time of our algorithms with the different implementations from both sides of theory and practice. The computational results show that the efficient implementations significantly reduced the running time of the bottom-left and the best-fit algorithms, and our algorithms are especially efficient for instances having repeated shapes. We also create large-scale instances with up to 10,000 distinct shapes to test our efficient algorithms and observe that the running time increases almost linearly with the number of distinct shapes.

We then analyze the strength and weakness of the bottom-left and the best-fit algorithms from the viewpoint of the quality of packing results. We investigate the reasons why the best-fit algorithm outperforms the bottom-left algorithm for many instances and situations when the bottom-left algorithm performs better for some kinds of instances. Based on these observations, we propose a new construction heuristic called the *partition-based best-fit heuristic* (*PBF*) as a bridge between the bottom-left and the best-fit algorithms. The basic idea of the PBF algorithm is that all the items to be packed are partitioned into groups, and then items are packed into the container in a group-by-group manner. The best-fit algorithm is taken as the internal tactics to pack items of each group. We show that the PBF algorithm has the same time complexity as the best-fit algorithm with similar implementations. We perform a series of experiments to compare the performance with respect to the occupation rates of the packing layouts obtained by the PBF algorithm and those obtained by the bottom-left and the best-fit algorithms. The computational results show that the PBF algorithm significantly improves the occupation rate of the packing layouts, and it is especially effective for instances with many different sizes of shapes.

As explained before, our algorithms may also be used to obtain good solutions in reasonable time for the problem of packing arbitrarily shaped polygons by representing each arbitrarily shaped polygon as a bitmap image. The techniques used in our algorithms may give insight to help design efficient implementations of algorithms for large-scale instances of combinatorial optimization problems.

This thesis is organized as follows. In Chapter 2, we survey some background of combinatorial optimization problems and packing problems. Then, in Chapter 3, we give the formulation of the rectilinear block strip packing problem considered in this thesis and several important definitions and techniques used in our algorithms. We explain our construction heuristics for the rectilinear block packing problem and their implementations from Chapter 4 to Chapter 6. In Chapter 7, we briefly explain how we can apply the efficient implementation to the rectilinear block packing problem with rotation. We show that the time complexity of the case with rotation is the same as that without rotation. Finally, in Chapter 8, we summarize our results in this thesis.

Chapter 2

Background

In this Chapter, we survey some background of combinatorial optimization problems and packing problems. We first give some representative combinatorial optimization problems and their complexities in Sections 2.1 and 2.2. Then, we explain the concept and some techniques of approximation and heuristic algorithms in Section 2.3. We survey some background of packing problems in Section 2.4 and then explain some previous works for rectilinear block packing problems in Section 2.5.

2.1 Combinatorial optimization problems

Combinatorial optimization [63] is one of the most active areas in operations research, computer science and discrete mathematics. It appears in various fields such as operations research, applied mathematics and theoretical computer science. Combinatorial optimization problems involves various applications, such as airline crew scheduling, VLSI design and network design. A variety of real-life problems can be abstracted as combinatorial optimization problems.

In general, combinatorial optimization is a problem that involves finding an optimal object from a finite set of objects. The *job assignment problem* that usually appears in real-world applications is given as a simple example of combinatorial optimization problems. Given a set of jobs to be done and a set of employees, we are asked to assign employees to jobs and the objective is to make all jobs done as early as possible. Each job j has a processing time t_j and can be done by a subset of the employees. We assume that every processing time t_j is in $\mathbb{Z}^+$, where $\mathbb{Z}^+$ is the set of nonnegative integers $\{0, 1, 2, \ldots\}$, and all employees are equally efficient, i.e., it takes the same amount of time to complete a job for any employee who can do the job. We can assign several employees to the same job at the same time, and one employee to several jobs (not at the same time). This problem

can be formally described as follows:

Job Assignment Problem

Instance:

A set of processing times $t_1, t_2, \ldots, t_n \in \mathbb{Z}^+$ for n jobs, $m \in \mathbb{Z}^+$ employees, and a nonempty subset $S_j \subseteq \{1, 2, \ldots, m\}$ for each job $j \in \{1, 2, \ldots, n\}$.

Task:

Find a solution $x_{ij} \in \mathbb{Z}^+$ for all $i \in S_j$ and $j = 1, \ldots, n$ such that $\sum_{i \in S_j} x_{ij} = t_j$ for every j and $\max_{i \in \{1,2,\ldots,m\}} \sum_{j:i \in S_j} x_{ij}$ is minimum.

There will be many *feasible solutions* for this problem, where a solution is called feasible if it satisfies all constraints. A solution x for this problem is feasible if it satisfies $\sum_{i \in S_j} x_{ij} = t_j$ for every j, i.e., the total amount of work time assigned to job j equals the processing time t_j of the job. Because no employee can contribute to different jobs at the same time, the time when all jobs are done equals the maximum total working time assigned to an employee, i.e., $\max_{i \in \{1,2,\ldots,m\}} \sum_{j:i \in S_j} x_{ij}$. Hence the objective of this problem is in fact to minimize the maximum total working time of employees. A feasible solution that minimizes this value is called an *optimal solution*, and its objective value is called the *optimal value*. Theoretically, an optimal solution can be found by completely enumerating all feasible solutions. However, in most cases, such enumeration is impractical even when the input size of the problem is not large because the number of all feasible solutions may increase exponentially with the input size.

An algorithm is regarded as "good" or "efficient" if it runs in polynomial time. This concept was introduced by Edmonds [30]. Some of the classical problems in combinatorial optimization have polynomial time algorithms. Two examples of such problems are given in the following.

Minimum Spanning Tree Problem

Instance:

An undirected graph $G = (V, E)$ and weights on the edges $w : E \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers.

Task:

Find a spanning tree in G having the minimum total edge weight.

There are two famous polynomial time algorithms for the minimum spanning tree problem, *Kruskal's algorithm* [64] and *Prim's algorithm* [85]. Kruskal's algorithm works in $O(|E| \log |V|)$ time using a disjoint-set data structure (Union and Find). Prim's algorithm

implemented with a balanced search tree or a binary heap runs in $O(|E| \log |V|)$ time in the worst case. Implementations for Prim's algorithm with Fibonacci heap can improve its running time to $O(|E| + |V| \log |V|)$.

Shortest Path Problem

Instance:

A graph $G = (V, E)$, two vertices $s, t \in V$, and weights on the edges $w : E \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers.

Task:

Find a shortest s-t path P having the minimum total edge weight.

There are also two famous polynomial time algorithms for the shortest path problem, the *Bellman-Ford algorithm* [13] and *Dijkstra's algorithm* [26]. The Bellman-Ford algorithm can solve this problem and find shortest paths from s to all $v \in V$ in $O(|V||E|)$ time. Dijkstra's algorithm works in $O(|E| \log |V|)$ time when the weights on edges are nonnegative, i.e., $w : E \to \mathbb{R}^+$ where $\mathbb{R}^+$ is the set of nonnegative real numbers. This running time can be improved to $O(|E| + |V| \log |V|)$ if implemented with Fibonacci heap [31].

2.2 NP-hardness

Most of complexity theory is based on *decision problems* that asks to answer "yes" or "no" [34]. We can derive decision problems from minimization (resp., maximization) problems by adding a bound B as a parameter and asking whether there exists a solution with value not more than (resp., at least) B. For example, a decision problem derived from the shortest path problem can be described as follows:

Shortest Path Decision Problem

Instance:

A graph $G = (V, E)$, two vertices $s, t \in V$, weights on the edges $w : E \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers, and a bound B.

Task:

Does G contain an s-t path P whose total edge weight is not more than B.

The class P consists of all decision problems solvable in polynomial time. The decision problems derived from the minimum spanning tree problem and the shortest path problem are in P.

The class NP is defined to be the class of all decision problems that can be solved by polynomial time nondeterministic algorithms. Suppose for an instance of a decision problem in NP, one claims that the answer of the instance is "yes" by providing a certificate such as a solution. Then we can verify the certificate in polynomial time. Let us consider the following *3-satisfiability problem* (*3SAT*) as a simple example.

3-Satisfiability Problem

Instance:

A set X of variables and a collection $\mathcal{Z}$ of clauses over X, each consisting of exactly three distinct literals.

Task:

Decide whether $\mathcal{Z}$ is satisfiable.

This problem belongs to NP. It is not known whether there exists a polynomial time algorithm for solving this problem. However, if we are given an assignment of X for an instance of this problem as a certificate, it is obviously possible to verify a "yes" answer for the instance in polynomial time by checking whether the assignment is in fact a truth assignment satisfying $\mathcal{Z}$.

The class NP includes (the decision problem versions of) most combinatorial optimization problems, and it also contains all the problems in P, i.e., P $\subseteq$ NP. It is not known whether P = NP, which is one of the most important open problems in complexity theory.

A decision problem in NP is defined to be NP-*complete* if all other decision problems in NP can be reduced to it in polynomial time. This implies that if an NP-complete problem can be solved in polynomial time, then every problem in NP can be solved in polynomial time, i.e., P = NP. The 3SAT problem is known to be NP-complete and it is very useful when we prove another problem is also NP-complete: all we have to do is to prove that the 3SAT polynomially transforms to that problem.

The definition of NP-*hard* applies to a more general class of problems called *search problems*. For a search problem, we are given a subset of solutions, which is defined, e.g., by a set of constraints, and are asked to search for a solution in the subset or to return the answer "no" if such a solution does not exist. Such a subset can be the set of all optimal solutions of an instance, and hence all optimization problems belong to the class of search problems.

A search problem that is at least as hard as an NP-complete problem (i.e., all problems in NP can be polynomially reduced to it) is called NP-hard. NP-hard problems are not necessarily in NP, since they may not be decision problems.

We can say that a search problem is NP-hard if the corresponding decision problem is known to be NP-complete. The NP-completeness result for a decision problem can thus be translated into an NP-hardness result for its corresponding search problem. The following *maximum 3-satisfiability problem* (*MAX-3SAT*) is a search problem corresponding to 3SAT.

Maximum 3-Satisfiability Problem

Instance:

A set X of variables and a collection $\mathcal{Z}$ of clauses over X, each consisting of exactly three distinct literals.

Task:

Find a truth assignment that satisfies the largest number of clauses in $\mathcal{Z}$.

The MAX-3SAT is NP-hard, because 3SAT is NP-complete. Given an instance of a collection $\mathcal{Z}$ for 3SAT, which is also an instance for MAX-3SAT, determining whether $\mathcal{Z}$ is satisfiable or not for the 3SAT is obviously not harder than finding an optimal solution for the MAX-3SAT.

Some examples of NP-hard problems are given in the following. All of these problems are typical combinatorial optimization problems.

Traveling Salesman Problem

Instance:

A directed graph $G = (V, E)$ and weights on the edges $w : E \to \mathbb{R}^+$.

Task:

Find a Hamiltonian cycle whose total edge weight is minimum.

Set Covering Problem

Instance:

A finit set U, a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ where $\bigcup_{S_i \in \mathcal{S}} S_i = U$, and weights $w : \mathcal{S} \to \mathbb{R}^+$.

Task:

Find a subset $\mathcal{R} \subset \mathcal{S}$ such that $\bigcup_{S_j \in \mathcal{R}} S_j = U$ and $\sum_{S_j \in \mathcal{R}} w_j$ is minimum.

Vertex Coloring Problem

Instance:

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

Task:

Find a vertex coloring $f : V \to \{1, 2, \ldots, k\}$ of G with $f(v) \neq f(w)$ for all $\{v, w\} \in E$ and k is minimum.

It is strongly believed that NP-hard problems have no polynomial time algorithms [34]. Solving these problems exactly may necessitate enumerating an essential portion of all solutions that increases exponentially with the size of input. Representative methods such as *branch-and-bound* and *dynamic programming* are frequently utilized to find exact optimal solutions by enumerating only promising solutions efficiently [45]. Many algorithms are proposed for NP-hard problems utilizing these techniques. It is often the case that they can exactly solve some small-scale instances and some relatively large instances with simple or special structures. However, for the problems that have complex structures or for the large-scale instances, there are still no efficient algorithms to find their exact optimal solutions.

In many practical applications, however, it is not really necessary to find exact optimal solutions, and sufficiently good solutions would be enough. Hence, approximation and heuristic algorithms that compute sufficiently good solutions in reasonable time become very important. The concept and some techniques for approximation and heuristic algorithms are introduced in Section 2.3

2.3 Approximation and heuristic algorithms

In this section, we give the important concept and some typical techniques of approximation and heuristic algorithms. In principle, any algorithm that computes a feasible solution of a problem is an approximation or heuristic algorithm. The basis is to pay for the reduced running time by getting only a suboptimal solution.

Assume that for an instance of an optimization problem, z^A is the solution value computed by an approximation algorithm A and OPT is the optimal value. If $z^A/\mathrm{OPT} \leq \alpha$ holds for every instance of the problem, the algorithm A is called an *α-approximation algorithm* and α is called the approximation factor. Such approximation algorithms with performance guarantees have been investigated in the past two decades and the theory of hardness of approximation have been developed [8, 41, 90].

Techniques such as *greedy algorithm* and *local search* are usually considered when designing heuristics [1, 24]. These ideas are useful when devising heuristic algorithms for NP-hard problems. Algorithms derived from these techniques are often easy to implement and run in short time in practice.

A *greedy algorithm* [24] builds a solution by repeating simple steps, making a decision

at each step myopically to optimize some underlying criterion. It always makes a locally optimal choice, hoping that this choice will lead to a globally optimal solution. There usually exist many different greedy algorithms for the same problem. Greedy algorithms are guaranteed to find optimal solutions for some problems, and for many problems it can produce solutions that are guaranteed to be close to optimal. Greedy algorithms are quite powerful and work well for a wide range of problems. Many algorithms based on greedy method are proposed for combinatorial optimization problems, e.g., Kruskal's and Prim's algorithms for the minimum spanning tree problem, Dijkstra's algorithm for the shortest path problem explained in the previous section, and Chvatal's greedy heuristic for the set covering problem [22].

Local search [1] starts from an initial solution and repeats replacing the current solution with a better solution in its neighborhood until no better solution is found. It is a useful technique when designing heuristics algorithms, which explores the space of possible solutions sequentially. Although, in general, it is not a polynomial time method, it is observed by many researchers that it typically obtains near-optimal solutions in reasonable time for many problems.

Since simply applying local search only once in a heuristic algorithm may not be sufficient to find a good solution, many variants of simple local search such as multi-start local search (MLS), iterated local search (ILS), variable depth search are often considered when longer computation time is available. These variants are called metaheuristics in general.

The *MLS* [73], which is one of the simplest metaheuristics, first generates a series of initial solutions randomly and then applies local search for each initial solution independently. It finally returns the best solution among the resulting locally optimal solutions.

The *ILS* [68] iterates local search many times. For each iteration, it starts from an initial solution generated by slightly perturbing a good solution obtained so far. It is important for the performance of algorithm to generate initial solutions that retain some features of good solutions and care should be taken to avoid a cycling of solutions.

Recently, there have been an enormous number of studies on various types of *metaheuristics* [35, 40, 46, 80, 81, 91] such as tabu search, simulated annealing, genetic algorithms and greedy randomized adaptive search procedures (GRASP). When more quality is needed and more computation time is available, these techniques are often very effective and yield good results in practice. Osman and Laporte provides a classification of a comprehensive list of 1380 references on the theory and application of metaheuristics in [81]. Below we briefly introduce some representative metaheuristics.

The *tabu search* [36] tries to enhance local search by using the memory of previous searches. It repeatedly replaces the current solution with its best neighbor that is not

included in a list, called *tabu list*, that records recently visited solutions.

The *simulated annealing* [60] is a kind of probabilistic local search in which test solutions are randomly chosen from its neighborhood and accepted with a probability that equals 1 if the test solution is better than the current solution and is positive even if it is worse than the current solution. The acceptance probability of moves is controlled by a parameter called *temperature*. This idea drives from the physical process of annealing.

The *genetic algorithm* [42] is inspired by the evolutionary process in nature. It repeatedly generates a set of new solutions by applying two types of operations called *crossover* and *mutation* to the set of current solutions. The crossover operation generates one or more new solutions by combining two or more current solutions. The mutation operation generates a new solution by slightly perturbing a current solution.

The *GRASP* [86] usually consists of a greedy randomized construction procedure and a local search. It iterates such a construction procedure and local search many times. In the construction phase, it generates a feasible solution that is used as the initial solution in the local search phase.

2.4 Packing problems

In this section, we survey some background of packing problems. Packing problems belong to a class of optimization problems. In a packing problem, it involves packing a set of objects, called items, into containers. The objective is to pack a single container as densely as possible, or pack all items into as few containers as possible. A generic form of the packing problem is as follows:

Packing Problem

Instance:

A set of n items, and a set of containers.

Task:

Pack the items into containers without overlap so as to minimize or maximize a given objective function.

There are variety of objective functions for packing problems. Some typical objective functions are shown as follows:

- minimize the size of one container, e.g., strip packing problem,
- minimize the number of used containers, e.g., bin packing problem,

- maximize the total profit of the packed items, e.g., knapsack problem.

Many of these problems are related to real-world applications and they have been studied for a long time from both theoretical and practical points of view. There are many survey papers for these problems such as the typologies given by Dyckhoff [29] and Wäscher et al. [92], and a review by Hopper and Turton [44]. We explain some typical packing problems according to their dimensions in the following sections.

We first explain the simplest version of the packing problems, the one-dimensional packing problem. There are two major problems, the knapsack and the bin packing problem. These problems have simple structures. Many simple greedy algorithms performs well for these problems and many exact algorithms based on branch-and-bound or dynamic programming have also been proposed. However, packing problems with more than one dimension become much more complex. It seems very hard to find optimal solutions even for small instances. In this thesis, we focus on the two-dimensional strip packing problem. We give some background of the rectangle strip packing problem and the irregular strip packing problem. Finally, we briefly explain the three-dimensional packing problem.

2.4.1 One-dimensional packing problems

One-dimensional packing is the most basic category of packing problems, and many algorithms have been proposed. There are two major problems, the knapsack problem and the bin packing problem, in this category.

Knapsack problem

The *knapsack problem* [58] is one of the representative combinatorial optimization problems. In the knapsack problem, we are given a set of items and a knapsack with a specified capacity. The task is to select a subset of items such that the total profit of the selected items is maximized and the total weight does not exceed the given capacity. This problem has been studied for decades since it is the simplest prototype of maximization problems. The knapsack problem is formally described as follows:

Knapsack Problem

Instance:

Sets of profits $\{p_1, p_2, \ldots, p_n\}$ and weights $\{w_1, w_2, \ldots, w_n\}$ for n items, and a capacity value c. (Usually, all the values are positive integer numbers.)

Task:

Find a subset $S \subset \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq c$ and $\sum_{i \in S} p_i$ is maximum.

There are many simple greedy algorithms proposed to solve the knapsack problem and most of them perform well in practice. An intuitive idea for greedy algorithms for the knapsack problem is to consider the *efficiency* $e_i = p_i/w_i$ of each item i and try to select the items with high efficiency into the knapsack. Such items contribute large profit while consuming small amount of capacity. We first sort the items in decreasing order of their efficiency such that $e_1 \geq e_2 \geq \cdots \geq e_n$. The greedy algorithm starts with an empty knapsack $S := \emptyset$ and go through the items in this decreasing order putting every item j into S if $\sum_{i \in S} w_i + w_j \leq c$ holds.

Dantzig [25] proposed a linear programming relaxation (LP-relaxation) of the original problem, which is derived using real numbers instead of only integer values. Assuming OPT is the optimal value of an instance of the knapsack problem, it is obvious for a maximization problem that the optimal value z^{LP} of the relaxed problem is at least as large as the optimal value of the original problem, i.e., $z^{LP} \geq \mathrm{OPT}$. We also assume that the items are already sorted according to their efficiency. The solution of LP-relaxation can be computed similarly to the above greedy algorithm. It also starts with an empty knapsack and go through the items in the decreasing order putting items into the knapsack. We call an item s a *split item* if $\sum_{i=1}^{s-1} w_i \leq c$ and $\sum_{i=1}^{s} w_i > c$ holds. The algorithm returns a solution consisting of the first $s-1$ items and an appropriate fractional part $(= (c - \sum_{i=1}^{s-1} w_i)/w_s)$ of item s. The optimal value z^{LP} of the LP-relaxation is $\sum_{i=1}^{s-1} p_i + ((c - \sum_{i=1}^{s-1} w_i)/w_s) \cdot p_s$, which is called *Dantzig bound*.

Note that $\mathrm{OPT} \leq z^{LP} \leq \sum_{i=1}^{s-1} p_i + p_s$ holds. Another greedy algorithm is to compare the profit of the solution consisting of the first $s-1$ items and that of the solution with a single item having the highest profit. The algorithm adopts the solution with larger profit $(= \max\{\sum_{i=1}^{s-1} p_i, \max\{p_1, p_2, \ldots, p_n\}\})$. Assuming that z is the total profit value of items in the solution returned by this greedy algorithm, $\mathrm{OPT} \leq z^{LP} \leq \sum_{i=1}^{s-1} p_i + p_s \leq 2z$ holds. Hence, this greedy algorithm is a $\frac{1}{2}$-approximation algorithm.

There are also pseudo-polynomial time algorithms using dynamic programming [12,25] and fully polynomial time approximation schemes (FPTAS) for this problem. The first FPTAS for the knapsack problem was proposed by Ibarra and Kim [47]. The FPTAS with the currently known best complexity, both for the running time and the space, was proposed by Kellerer and Pferschy in [57].

From practical experience it is known that many knapsack problem instances of considerable size can be solved in reasonable time by exact algorithms using the techniques

of dynamic programming and branch-and-bound [58].

Balas and Zemel [10] gave the precise definition of the core of a knapsack problem based on the knowledge of an optimal solution. Some algorithms are based on *primal-dual dynamic programming recursions*, the concept of *solving a core* and the *separation of cover inequalities* [83, 84] to tighten the formulation. Martello et al. gave an overview of the latest techniques of exact algorithms for the knapsack problem in [72].

Bin packing problem

In the *bin packing problem*, we are given a set of items and bins of equal capacity and the task is to assign all the items to the bins such that the total size of the items assigned to each bin does not exceed its capacity and the number of the used bins is minimum. Without loss of generality, the capacity of each bin is assumed as 1. The bin packing problem is formally described as follows:

Bin Packing Problem

Instance:

A set of items and their sizes of nonnegative numbers $a_1, a_2, \ldots, a_n \leq 1$.

Task:

Find a $k \in \mathbb{Z}^+$ and an assignment $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, k\}$ such that $\sum_{i:f(i)=j} a_i \leq 1$ for every $j \in \{1, 2, \ldots, k\}$ and k is minimum.

This problem is equivalent to the simplest version of the *cutting stock problem* [41]. A generic form of the cutting stock problem is described as follows:

Cutting Stock Problem

Instance:

A set of n items, each with size $a_i \in \mathbb{Z}^+$ and demand $d_i \in \mathbb{Z}^+$, and beams of equal length l.

Task:

Cut beams into pieces such that the total size of pieces from each beam is not larger than l, and for each $i = 1, 2, \ldots, n$, we have d_i or more pieces of length a_i so as to minimize the number of used beams.

The simplest version of the above cutting stock problem is to set every demand d_i to one and the resulting problem is equivalent to the bin packing problem.

The bin packing problem is proved to be NP-hard in the strong sense and no algorithm can achieve a performance ratio strictly better than 3/2 unless P = NP. Below we

introduce three well-known greedy heuristic algorithms for this problem. Let us denote the optimal value of an instance of this problem by OPT.

The simplest heuristic algorithm among the three greedy algorithms is called the *next-fit* algorithm [63]. It assigns items one by one into bins. Starting from one bin, the algorithm assigns an unassigned item to the bin most recently opened if it has sufficient remaining capacity, and whenever the capacity of the current bin would exceed after assigning the item, the algorithm closes the current bin, opens a new bin, and then assigns the item to the new bin. Assuming that NF is the number of bins that is returned by the next-fit algorithm, the next-fit algorithm runs in $O(n)$ time and for any instance, it satisfies $\mathrm{NF} \leq 2\mathrm{OPT} - 1$ (see the proof in [63]). Hence, the next-fit algorithm is a 2-approximation algorithm. Furthermore, there exists an instance which shows that this bound is tight. It consists of $2n$ items of sizes $\{1/2, 1/n, 1/2, 1/n, \ldots, 1/2, 1/n\}$. The OPT of this instance is $n/2 + 1$, and the value of the solution returned by the next-fit algorithm is n.

The *first-fit* algorithm [63] is similar to the next-fit algorithm. At the beginning, it prepares n bins. It assigns items one by one to bins where each item is assigned to the oldest bin (having the smallest id) whose residual capacity is not smaller than the size of the item. The first-fit algorithm is also a 2-approximation algorithm. Assuming that FF is the number of bins that is returned by the first-fit algorithm, $\mathrm{FF} \leq \lceil \frac{17}{10}\mathrm{OPT} \rceil$ holds [54]. There exist instances with arbitrarily large OPT and $\mathrm{FF} \geq \frac{17}{10}(\mathrm{OPT} - 1)$.

Analysis shows that the next-fit and the first-fit algorithms perform well when the sizes of items are small. It is proved that $\mathrm{NF} \leq \lceil \sum_{i=1}^{n} a_i/(1-\gamma) \rceil$ holds for any γ that satisfies $0 < \gamma < 1$ and $\gamma \geq a_i$ for every i. It is natural to assign larger items first. The *first-fit-decreasing* algorithm [63] first sorts the items in decreasing order of their sizes and then applies the first-fit algorithm to assign them to bins. The first-fit-decreasing algorithm is a $\frac{3}{2}$-approximation algorithm [88] and it is shown that $\mathrm{FFD} \leq \frac{11}{9}\mathrm{OPT} + \frac{2}{3}$ holds for any instance, where FFD is the number of bins used by the first-fit-decreasing algorithm [27].

2.4.2 Two-dimensional packing problems

Two-dimensional packing problem involves packing a set of two-dimensional items into larger rectangular containers such that no item overlaps with others. Items in the problem can be shaped arbitrarily such as the rectangles, circles and irregular shaped polygons. As explained at the beginning of this section, there are many variations for the objective functions. Among those, the *strip packing problem* has been intensively investigated. In the strip packing problem, we are given a set of polygons, rectangles or irregular shaped polygons, and only one rectangular container whose width is fixed and height is a variable. The objective is to minimize the height of the given container. This problem is categorized

as *open dimension problem* in a recent typology proposed by Wäscher et al. [92]. When all the given items are rectangles, this problem is called the *two-dimensional strip packing problem* (*2SP*) [92] and that for polygons is called the *irregular strip packing problem*.

There are some typical strategies and algorithms proposed for the strip packing problem. Hopper and Turton [44] gave a review of metaheuristics developed to two-dimensional packing problems. The *bottom-left* strategy [9,43] and the *clustering* strategy [2] are known as the most popular strategies for this problem. The bottom-left strategy, which was originally proposed for the rectangle packing problem [9], packs items into the container one by one at the lowest position as left as possible. Figure 2.1 shows an example of packing items according to the bottom-left strategy. The clustering strategy always selects a subset of the given items and packs them as compactly as possible and then treats the resulting combination of items as a new item. There are also some heuristic algorithms based on clustering strategy proposed in the literature [14, 28, 52, 53]. Figure 2.2 shows an example of the processing when packing items according to the clustering strategy.

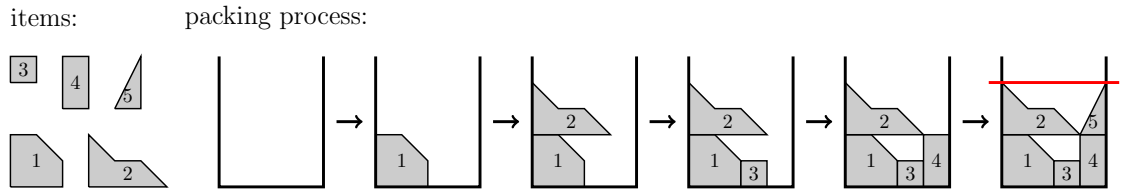


Figure 2.1: Bottom-left strategy

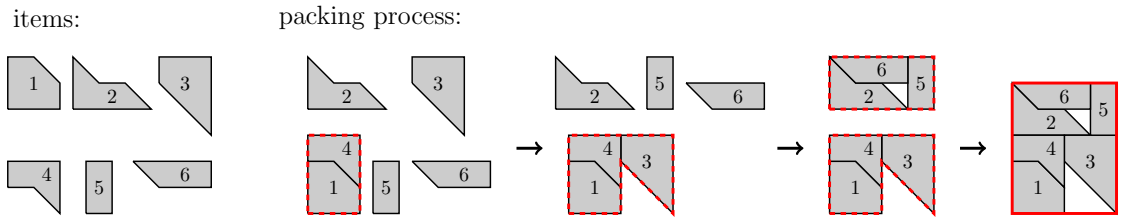


Figure 2.2: Clustering strategy

The 2SP has various applications in material industry and scheduling problems. There are many heuristic algorithms based on the explained strategies in the literature [9, 43]. As explained, the bottom-left strategy was first proposed for the 2SP in [9] and the algorithm is called the *bottom-left algorithm*. The bottom-left algorithm is one of the simplest forms among the algorithms based on the bottom-left strategy. Burke et al. [19] proposed another algorithm called the best-fit algorithm, which is slightly more complicated than the bottom-left algorithm. The research in this dissertation is inspired by these two algo-

rithms and we explain them in details in Chapter 3. There are also some heuristics based on local search [50, 51] for the 2SP.

Since rectangle is the simplest shape of polygons and it is easy to define the relative positions among them, there are some exact algorithms [59, 65, 71] proposed for the 2SP utilizing this special structure. Alvarez-Valdes et al. [6] proposed new lower bounds for 2SP and a branch-and-bound algorithm obtained by incorporating a GRASP algorithm and an exact algorithm based on staircase placement.

The irregular strip packing problem also has numerous material industry applications such as paper and textile industries. The algorithm proposed in [7] utilizes the concept of the bottom-left strategy. It first approximates every non-convex polygon by a convex polygon that encloses it and then places them one by one at a feasible bottom-left position. Heuristics [18, 20] also takes the bottom-left strategy when packing items. Recall that the bottom-left algorithm packs items one by one according to a given sequence. Many approaches have been proposed for finding a good sequence in [5, 37, 79]. Adamowicz and Albano [3] proposed an algorithm using the concept of the clustering strategy. Their algorithm first partitions the items into several groups, and then generates a rectangle enclosure for each group where the polygons in each group are packed as compactly as possible.

Algorithms that take advantages of mathematical programming techniques for the irregular strip packing problem are proposed in [15, 38, 66]. Benell and Dowsland [15] proposed an algorithm that incorporates both the bottom-left algorithm and a linear programming based *compaction algorithm*, where a compaction algorithm translates the packed items continuously so as to minimize the height of the container. There also exists a *separation algorithm* [66], which is often used in combination with a compaction algorithm. The separation algorithm works for a packing layout where there are some polygons having overlap with others. It translates the items in the layout continuously in order to make them separate with each other. The algorithm proposed in [38] updated many best known results for the benchmark instances of the irregular strip packing problem. It featured the bottom-left algorithm with the linear programming based compaction and separation algorithms and it was further incorporated with simulated annealing.

2.4.3 Three-dimensional packing problems

Three-dimensional packing problems appear in various industrial applications and there are some variations of this problem such as container loading problem, three-dimensional strip packing problem, three-dimensional bin packing problem and three-dimensional knapsack problem. These variations have similar properties as those of the one- or two-

dimensional packing problems. The three-dimensional strip packing problem involves packing a set of cuboid items into one container with fixed basal shape and infinite height so as to minimize the height of the container. The three-dimensional bin packing problem asks to pack given items into bins whose shapes and sizes are given as input and the objective is to minimize the number of used bins. In the three-dimensional knapsack problem, we are given a set of cuboid items, each with a profit, and a container, and the task is to select a subset of items such that they can be packed inside of the container without overlap and the total profit is maximum.

There are many heuristic algorithms proposed for these problems. Most of them are based on heuristic algorithms proposed for the one- and two-dimensional packing problems. Bischoff and Marriott in [16] introduced 14 heuristics for the three-dimensional strip packing problem and also performed a series of experiments to compare the performance of these algorithms.

Note that in real-world applications, there exist many complicated constraints needed to be considered such as the compactness of one container, the center of gravity, stability of the packing layout, etc. Bortfeldt and Wäscher [17] provide an overview of real-world container loading problems. They pointed out that many algorithms in the literature are of limited practical value since they do not pay enough attention to constraints encountered in practice.

2.5 Rectilinear block packing problems

Rectilinear block packing problem involves packing a set of arbitrarily shaped rectilinear blocks into a larger rectangular container without overlap so as to minimize or maximize a given objective function. A rectilinear block is a polygonal block whose interior angle is either 90° or 270°. This problem involves many industrial applications, such as VLSI design, timber/glass cutting, and newspaper layout. It is among classical packing problems and is known to be NP-hard [9].

It is difficult to represent the relationships among rectilinear blocks than those among rectangles. Several structures have been designed to represent the relationships among rectilinear blocks such as the *bounded-sliceline grid* (*BSG*) [78, 87], *sequence-pairs* [32, 33, 56, 94], O-tree [82], B*-tree [93], *transitive closure graph* (*TCG*) [67] and *corner block list* (*CBL*) [69]. Several heuristics have been proposed for the rectilinear block packing problem based on these structures.

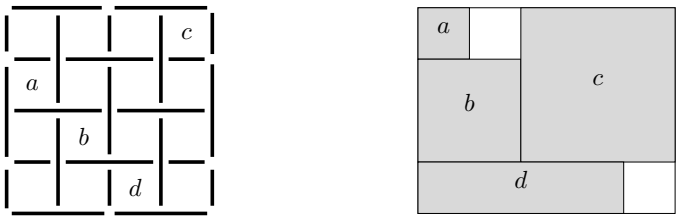
We explain two representative techniques, the BSG and the sequence-pair representations, in the following subsections. Both of them are often used for the two-dimensional packing problem whose objective is to minimize the total area containing all the items,

called the *area minimization problem*. The methods based on these representations perform well for small-scale instances in reasonable running time, and are especially effective for instances consisting of rectilinear blocks that have simple shapes (e.g., L- or T-shape).

2.5.1 Bounded-sliceline grid

Nakatake et al. proposed in [77] a *bounded-sliceline grid* (*BSG*) structure to represent the placement of items.

BSG is a topology defined in the plane and orthogonal line segments, called *BSG-segs*, dissect the plane into square zones, called *BSG-rooms* (see Figure 2.3 (a) as an example). The items are assigned to distinct BSG-rooms and they have the same relationship on the placement with that among the BSG-rooms. A BSG of size $n \times n$ with assignment of items to BSG-rooms can represent the relationship among given n items in a placement of the items. Figure 2.3 (b) shows a placement of four items according to the assignment in a BSG of size 4×4 in Figure 2.3 (a).



(a) (b)

Figure 2.3: An example of the BSG representation

Many approaches for the area minimization problem based on BSG are proposed in the literature. It is not guaranteed that there is a corresponding BSG representation for every placement for instances of L-shaped blocks packing problem. Because of this limitation, optimal solutions may not be included in the search space of algorithms based on BSG.

The algorithms for L-shaped blocks packing problem was proposed in [55, 77]. They partition each L-shaped block to two rectangles and assign them into two adjacent BSG-rooms. Since adjacent BSG-rooms have a common BSG-seg, it is easy to align the rectangles into the original shapes. These approaches may still miss the optimal solution.

In [78], a method for packing rectilinear blocks and blocks with positional constraints was proposed. The algorithm divides rectilinear blocks into a set of rectangles and packs them based on BSG. Because of the limitation of BSG representation, it may also miss

the optimal solution.

Although the above algorithms may miss optimal solutions, their performance is often good in practice.

Sakanushi et al. proposed in [87] an algorithm for packing convex rectilinear blocks based on multi-BSG, which is an arrangement of plural BSGs on multiple layers. They also represent a rectilinear block as a set of rectangles. The algorithm guarantees that an optimal placement is contained in its solution space.

2.5.2 Sequence-pair

The *sequence-pair* [74,75] was proposed in 1995 as a coding scheme to represent a packing layout of rectangles. The basic idea is that any feasible packing layout can be represented by fixing the relative positions of all the rectangles. This technique is also used to represent the solution space of the area minimization problem.

A sequence-pair represents a solution of δ rectangles by a pair of permutations of their names. The number of all possible sequence-pairs for δ rectangles is $(\delta!)^2$. A sequence-pair describes the relative horizontal or vertical positions for each pair of rectangles as follows:

- If $(\cdots a \cdots b \cdots, \cdots a \cdots b \cdots)$ holds, then a is left to b (i.e., the right edge of a is left to the left edge of b).
- If $(\cdots b \cdots a \cdots, \cdots a \cdots b \cdots)$ holds, then a is below b (i.e., the top edge of a is below the bottom edge of b).

For a simple example, if we have three rectangles named a, b and c, (abc, bac) is a sequence-pair that represents their packing layout. The sequence-pair (abc, bac) means that a is left to c, b is below a and b is left to c. Figure 2.4 shows a placement with minimum area for sequence-pair (abc, bac).

Sequence-pair: (abc, bac)

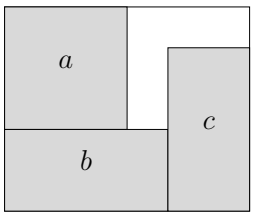


Figure 2.4: An example of an area minimum placement for a sequence-pair

One-dimensional compaction can be used to construct from a given sequence-pair a placement with minimum area. It greedily pushes every rectangle to the bottom left corner of the container and it can obtain such a placement in $O(\delta^2)$ time.

Fujiyoshi and Murata [33] generalized this technique to solve the rectilinear block packing problem. They deal with the packing problem whose objective is to minimize the area of the container. They first partition each rectilinear block as a set of rectangles since a sequence-pair can only deal with rectangles. Procedure called X/Y-alignment was proposed to align such rectangles. The algorithm to obtain a rectilinear block packing layout from a given sequence-pair, i.e., a decoding algorithm to obtain a feasible packing layout, was also proposed in [33]. The time complexity of the decoding algorithm is $O(m^3)$, where m is the total number of rectangles representing all the rectilinear blocks.

Machida and Fujiyoshi [70] (in Japanese) proposed an improved decoding algorithm that runs in $O(m^2 + n^3)$ where m is the total number of rectangles and n is the number of rectilinear blocks.

In a sequence-pair, if rectangles a, b, c, d are assigned in the order of

- $(\cdots a \cdots bc \cdots d \cdots, \cdots c \cdots ad \cdots b \cdots)$ or
- $(\cdots a \cdots bc \cdots d \cdots, \cdots b \cdots da \cdots c \cdots)$,

rectangles a, b, c, d make an *adjacent cross* [76]. There are at most $\lceil(\delta-2)/2\rceil\,\lfloor(\delta-2)/2\rfloor$ adjacent crosses in a sequence-pair with δ elements.

An efficient coding scheme for rectangle packing called *Selected Sequence-Pair* [62] is defined as a sequence-pair with k adjacent crosses. Any arbitrary rectangle packing layout can be represented by a selected sequence-pair and a packing layout with minimum area for a given selected sequence-pair can be obtained in $O(\delta + k)$ time. Takashima and Murata [89] proved that the necessary number of adjacent crosses for representing an arbitrary placement of δ rectangles is at most $\delta - \lfloor\sqrt{4\delta - 1}\rfloor$ i.e., $k \leq \delta - \lfloor\sqrt{4\delta - 1}\rfloor$. Hence, a given selected sequence pair can be decoded in linear time of the number of rectangles.

In [32], a decoding algorithm whose running time is $O((p+1)m)$ was proposed for a given selected sequence-pair for rectilinear blocks, where m also denotes the total number of rectangles representing all the rectilinear blocks and p is the number of rectilinear blocks that are represented by more than one rectangle (not shaped as rectangles).

Chapter 3

Formulation and Important Techniques

In this chapter, we first formally describe the rectilinear block strip packing problem considered in this thesis and then explain some definitions and several important techniques used in our algorithms. First, we give the formulation of our problem and important definitions that are used through this thesis in Section 3.1. We then introduce in Section 3.2 some sophisticated data structures that are utilized when implementing our algorithms. These data structures are explained in general form. We explain in Section 3.3 the concept of a crucial technique called no-fit polygon that is very useful when determining whether two polygons overlap each other in two-dimensional space. We use this technique in our algorithms to check whether two blocks have intersections. We take the bottom-left strategy, which was mentioned in Section 2.4.2, as the main strategy of our construction algorithms. In this strategy, starting from an empty layout, items are packed into the container one by one, and whenever a new item is packed into the container, it is placed at the BL position relative to the current layout. The definition of BL positions in general and an efficient algorithm, called Find2D-BL, to calculate the BL position for a rectangle are explained in Section 3.4 and 3.5. Finally, we introduce in Section 3.6 two representative construction heuristics for the rectangle packing problem, the bottom-left and the best-fit algorithms.

3.1 Formulation

In this section, we give the formulation of the rectilinear block packing problem and some definitions used through this thesis.

For the rectilinear block packing problem we consider in this thesis, we are given a set of n items of rectilinear blocks, and a rectangular container C (also called a strip) with

fixed width W and unrestricted height H. The task is to pack all the items orthogonally without overlap into the container. The rectilinear block packing problem is formally described as follows.

Rectilinear Block Packing Problem

Instance:

A set of n items $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of rectilinear blocks, each with a deterministic shape and size in $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$, and a rectangular container C with fixed width W and unrestricted height H.

Task:

Pack all of the items orthogonally without overlap into the container so as to minimize height H.

We assume that the bottom left corner of the container is located at the origin $O = (0, 0)$ with its four sides parallel to the x- or y-axis. The objective is to minimize height H of the container that is necessary to pack all of the given items. Note that the minimization of height H is equivalent to the maximization of the occupation rate defined by $\sum_{i=1}^{n} A(R_i)/WH$, where $A(R_i)$ denotes the area of a rectilinear shape R_i. As explained in Chapter 2, this type of problem is often called the strip packing problem (e.g., the rectangle strip packing problem (2SP) and the irregular strip packing problem for the case of irregularly shaped polygons). According to the improved typology of Wäscher et al. [92], strip packing problems are categorized into the two dimensional open dimension problem (2D ODP) with a single variable dimension.

We define the *bounding box* of an item R_i as the smallest rectangle that encloses R_i, and its width and height are denoted as w_i and h_i. We call the area of the bounding box, $w_i h_i$, the *bounding area* of R_i. The location of an item R_i is described by the coordinate (x_i, y_i) of its *reference point*, where the reference point is the bottom-left corner of its bounding box. For convenience, each rectilinear block and the container C are regarded as the set of points (including both interior and boundary points) whose coordinates are determined from the origin $O = (0, 0)$.

Let $R_i(x_i, y_i)$ be the rectilinear block R_i placed at (x_i, y_i), i.e., the region occupied by R_i when its reference point is located at (x_i, y_i). The region of $R_i(x_i, y_i)$ can be represented by *Minkowski sum*. The Minkowski sum of two sets $A \subset \mathbb{R}^2$ and $B \subset \mathbb{R}^2$ is defined as

$$A \oplus B = \{a + b \mid a \in A, b \in B\}, \tag{3.1.1}$$

where $a + b$ is the vector sum of a and b and R is the set of real numbers. For convenience, when B consists of a single point b (i.e., $B = \{b\}$), $A \oplus \{b\}$ is denoted as $A \oplus b$. Then, a

rectilinear block R_i placed at $v_i = (x_i, y_i)$ is represented by $R_i(x_i, y_i) = R_i \oplus v_i = \{p + v_i \mid p \in R_i\}$.

For a rectilinear block R_i, let $\mathrm{int}(R_i)$ be the interior of R_i (n.b., the boundary of R_i is not included in $int(R_i)$). Then the rectilinear block packing problem is formulated as follows:

$$\begin{aligned}
&\text{minimize} && H \\
&\text{subject to} && 0 \le x_i \le W - w_i, \quad 1 \le i \le n && (3.1.2)\\
& && 0 \le y_i \le H - h_i, \quad 1 \le i \le n && (3.1.3)\\
& && \mathrm{int}(R_i(x_i, y_i)) \cap R_j(x_j, y_j) = \emptyset, \ i \ne j. && (3.1.4)
\end{aligned}$$

The constraints (3.1.2) and (3.1.3) require that all rectilinear blocks be packed inside the container. The constraint (3.1.4) ensures that there exists no item overlapping with others.

Two cases of this problem are often considered in the literature: (1) all the items are not allowed to be rotated, and (2) all the items can be rotated 90°, 180° or 270°. However, the case without rotations is assumed in this thesis unless otherwise stated, because it is easy to apply the results in this thesis to the case with rotations as discussed in Chapter 7.

We summarize some important notations and terminologies used through this thesis in Table 3.1. The column of "Notation/Terminology" shows the notations and terminologies and that of "Definition" shows the corresponding meanings.

Table 3.1: Notations and terminologies

Notation/Terminology	Definition
n	number of rectilinear blocks
C	container (strip)
W	width of the container
H	height of the container
$A(R)$	area of a polygon R
bounding box	the smallest rectangle that encloses a polygon
bounding area	area of the bounding box of a polygon
reference point	the bottom left corner of the bounding box of a polygon
$R(x, y)$	polygon R placed at (x, y)
$\oplus$	Minkowski sum
$\mathrm{int}(R)$	set of all points inside of a polygon R

3.2 Data structures

Algorithms with different implementations devised to solve the same problem often differ dramatically in their efficiency. The design and analysis of efficient data structures have long been recognized as important subjects in implementation. There are various data structures as explained in details in [24, 39].

In this section, we introduce some sophisticated data structures that are used to implement our algorithms. These data structures are very powerful to help construct efficient algorithms for a variety of problems. We explain these data structures in general form, but it is easy to apply them to the implementations of our algorithms proposed in later chapters.

We first explain two *search tree structures* in Section 3.2.1 and 3.2.2 and then the *heap* in Section 3.2.3. Search tree data structures support many operations to deal with elements, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE. We can use a search tree as a dictionary. Since we mainly utilize the operations of SEARCH, INSERT and DELETE through our implementations, we just explain these three operations for a search tree. The heap data structure is an array object that we can view as a nearly complete binary tree. Heaps provide improved performance when we only need the operations of INSERT and DELETE.

3.2.1 Binary search tree

In computer science, a *binary search tree* is a particular type of data structure to store objects in memory. A binary search tree is a rooted binary tree where each internal node in the tree has at most two subtrees, the left and right subtrees.

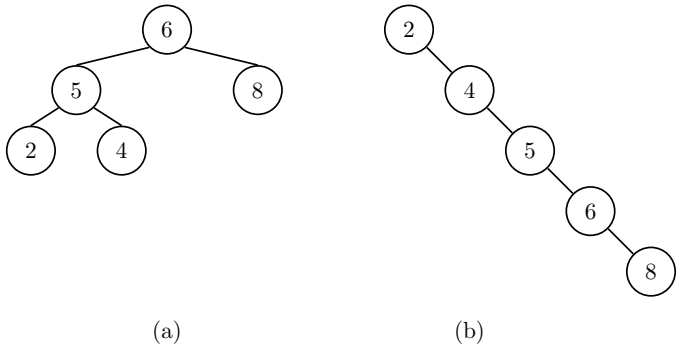


Figure 3.1: Examples of binary search trees in general

In general case, in a binary search tree, each node represents an object with a *key*. For every node v in the tree, the keys of nodes in the left subtree of v must not larger than that of v and similarly the keys of nodes in the right subtree of v must not smaller than that of v. Figure 3.1 illustrates two example binary search trees.

When we search for an object or a place to insert a new object into a tree, we traverse the tree from the root to a leaf according to the keys of nodes on the path. If the key that we are searching for is larger than the key of current node, we go down to its right child, otherwise go down to its left child. The running time for most of operations on a binary search tree is proportional to its height.

Different binary search trees can represent the same set of values, see Figure 3.1 as an example. The set of values $\{2, 4, 5, 6, 8\}$ can be represented by both trees shown in Figure 3.1 (a) and (b). In the worst-case, we may build a binary search tree with height $O(n)$ where n is the number of nodes in the tree (e.g., the tree in Figure 3.1 (b)). In particular, a search tree with height $O(\log n)$ is called a *balanced search tree*. In a balanced search tree, every path from any node on the tree to the root contains $O(\log n)$ nodes.

A *complete binary search tree* is a special binary search tree where all the leaves have the same depth. We consider the case when all the objects are assigned from left to right in decreasing order of their keys and the internal nodes are only used for navigation. As a result, a complete binary search tree has $2^{\lceil \log n \rceil}$ leaves and height $\lceil \log n \rceil$ to store n objects. This implies that there are $2^{\lceil \log n \rceil} - n$ dummy leaves in the tree. Figure 3.2 shows an example of the complete binary search tree. Each leaf node keeps a key x of an object and each internal node keeps a key (x, y) where x denotes the smallest value of the keys in its left subtree and y is the smallest value of the keys in its right subtree. The dummy leaves in the tree are marked by $*$.

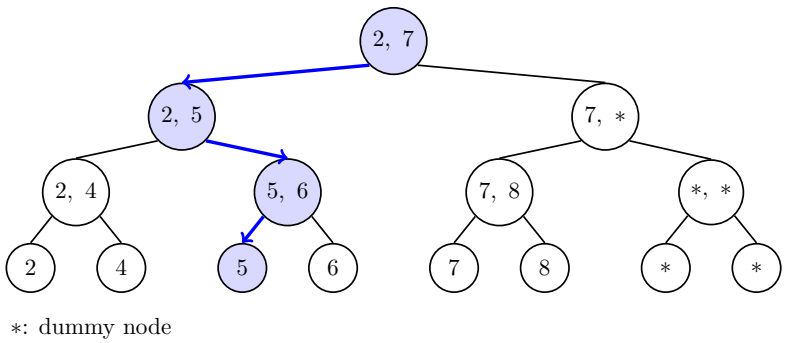


Figure 3.2: An example of a complete binary search tree

To build a complete binary search tree for n objects, we can first sort the objects in decreasing order of their keys. Then, we create the dummy leaves and internal nodes. Each internal node keeps the minimum values of the keys of its left and right children.

We can search for an object or a place for a new object with a key k by traveling the tree from the root to a leaf. Assuming that we arrive at an internal node v with key (x, y), there are three situations when we go down to a child of v as follows:

1. When $k < x$ holds, we return the place of the left side of the leftmost node in the tree rooted at v.

2. When $x \leq k < y$ holds, we go down to the left child of v.

3. When $k \geq y$ holds, we go down to the right child of v.

For example, we are asked to search for an object with key 5 in the tree shown in Figure 3.2. We begin the travel from the root with key $(2,\ 7)$. Since $2 \leq 5 < 7$ holds, we go down to the left child with key $(2,\ 5)$ and then go to its right child, the node with key $(5, 6)$. Finally we find the leaf that stores the object with key 5.

A binary search tree in which only leaves are used to store objects and internal nodes are used for navigation is sometimes easier to handle than a binary tree in which objects are stored in both leaves and internal nodes. The former only costs at most double space than the latter. The former style is utilized in the *Find2D-BL algorithm* [48], which is an algorithm to find the BL position for a rectangle and is explained in Section 3.5.

Building a complete binary search tree for storing n objects requires linear space and $O(n \log n)$ time. One can search for a specified value in a tree in $O(\log n)$ time. We omit the details of how to insert or to delete an object from a binary tree, because they are similar and simpler than those for a 2-3 tree, which are explained in the next section.

3.2.2 2-3 tree

We introduce a more advanced data structure called *2-3 tree*, which is a variant of *B-tree*. B-tree is a generalization of a binary search tree where an internal node can have more than two children [23, 61]. An interesting correspondence is known between B-trees and *red-black trees* [11, 24]. Many database systems use B-trees or variants of B-trees to store information.

The 2-3 tree, invented by Hopcroft [4], is a tree data structure where every internal node has either two or three children and all leaves have exactly the same depth. Internal nodes with two (three) children are called *2-nodes* (*3-nodes*). The children of an internal node are called left, middle and right child. A 2-node has only left and middle child. The

height of a 2-3 tree with n objects is between $\lfloor \log_3 n \rfloor$ and $\lfloor \log_2 n \rfloor$. Figure 3.3 shows a simple example of a 2-3 tree.

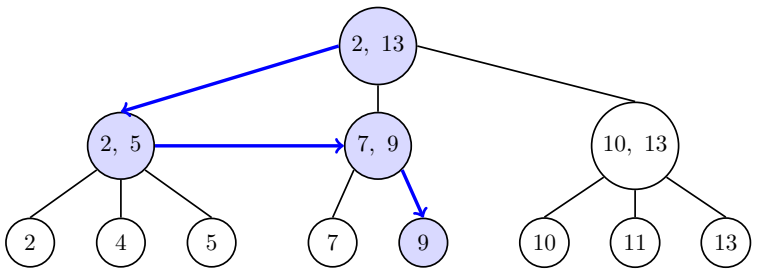


Figure 3.3: An example of a 2-3 tree

Generally, a 2-3 tree is a rooted tree with the following properties:

- All objects are stored in leaves. Internal nodes are only used for navigation.
- All leaves are at the same depth from the root.
- The root node can have two or three internal nodes or zero to three leaves as children.
- Every leaf node contains a single key x of the object stored in it.
- Every internal node has either two or three children and a key (x, y) where x is the smallest value and y is the largest value of the keys of its children.

The operations of SEARCH, INSERT and DELETE are explained in the following.

Procedure SEARCH on a 2-3 tree

One can search for a specified object or an appropriate place for a new object in a 2-3 tree by traveling the tree from the root downward to a leaf according to the keys of nodes on the path.

The procedure of SEARCH operation is formally described as 2-3-Tree-Search(v, k) in Algorithm 1. The 2-3-Tree-Search takes a pointer to the root v of a subtree and a key k to be searched for in that subtree as input. If an object with key k is in the subtree rooted at node v, 2-3-Tree-Search returns the corresponding leaf. Otherwise, the procedure returns a place for inserting that object.

Without loss generality, we assume that at the beginning $x \leq k \leq y$ holds, where (x, y) is the key of v. In other words, we assume that there must exist a leaf with key k or a

place to insert it between the leftmost and the rightmost leaves in the subtree rooted at v. If $k < x$ (resp., $k > y$), we can immediately return the place at the left (resp., right) side of the leftmost (resp., rightmost) leaf in the tree.

Algorithm 1 2-3-Tree-Search(v, k)

Input: The root v of a subtree and a key k.

Output: A leaf that contains key k or the place for inserting an object with key k.

1: If v is a leaf, return v.
2: Let (x, y) be the key of v, v_R be the sibling of v whose position is immediately to the right of v and v_l be the left child of v.
3: If $k < x$, return the place at the left side of the leftmost leaf in the subtree.
4: If $k > y$, return 2-3-Tree-Search(v_R, k).
5: If $x \leq k \leq y$, return 2-3-Tree-Search(v_l, k).

Figure 3.3 illustrates this procedure when we search for the key 9. The procedure examines the nodes on the path marked by arrows. One can call 2-3-Tree-Search$(root, k)$ at the beginning where *root* is the root of the 2-3 tree. Whenever the search goes downward from an internal node to one of its children, the procedure examines at most three nodes. Therefore, this SEARCH operation requires $O(\log n)$ time, where n is the number of nodes in the 2-3 tree.

Procedure INSERT on a 2-3 tree

The INSERT operation inserts a new object into a 2-3 tree. We first call the procedure 2-3-Tree-Search$(root, k)$ to find an appropriate place for the new object with key k. We then create a new leaf node α for the new object and insert it to the corresponding parent node v. If v was a 2-node before inserting α, we can obtain the resulting 2-3 tree just by updating the keys of nodes on the path from α to the root. If v was a 3-node before inserting α, i.e., it was full and have four children now, we need to split node v into two nodes so that each has only two children. If the parent of v was also full, we must also split it into two nodes and thus we need to split every 3-node and update the keys of the resulting nodes on the path from α to the root.

The process of splitting a 3-node is illustrated in Figure 3.4. We insert a subtree rooted at node α into the 3-node v and split v into two nodes v and u. We then insert nodes v and u to the root p and as a result p has four children. We split p into two nodes p and q and create a new node n to be the new root. The height of the tree thus grows by one. Note that splitting the root of a tree is the only case that the height of a tree grows.

The procedure of INSERT operation is formally described as 2-3-Tree-Insert(T, γ, α) in

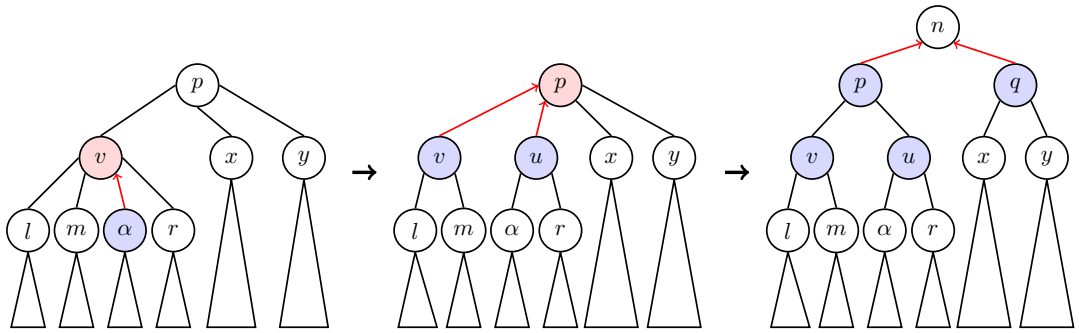


Figure 3.4: An example of splitting a 3-node in a 2-3 tree

Algorithm 2. The 2-3-Tree-Insert(T, γ, α) inserts a subtree rooted at node α at the place γ into 2-3 tree T.

If we are asked to insert an object having key k into 2-3 tree T, we first call procedure 2-3-Tree-Search$(root, k)$ to find the place γ for the new object, where *root* is the root of T. Then we create a leaf node α for the new object and call procedure 2-3-Tree-Insert(T, γ, α) to insert α into T.

The INSERT operation updates the keys of nodes and splits 3-nodes on the path from a leaf to the root without going down. Hence, it requires $O(\log n)$ time.

Algorithm 2 2-3-Tree-Insert(T, γ, α)

Input: A tree T, a subtree rooted at α and a place γ in T.

Output: Tree T after inserting the subtree rooted at α at γ.

1: Let *root* be the root of T.
2: Insert α at γ to T. Let v be the parent of α.
3: Update the key of v.
4: If v has four children, go to Step 5; otherwise, go to Step 7.
5: If v is the root, create a new root node n to be the parent of v and let $root := n$.
6: Let p be the parent of v. Split v into two nodes v_1 and v_2, each with two children. Set the keys for v_1 and v_2 and connect them to be children of p. Set $v := p$ and return to Step 3.
7: Update the keys of nodes on the path from v to *root* and stop.

Procedure DELETE on a 2-3 tree

The DELETE operation removes an object from a 2-3 tree. It first searches for the leaf α that stores the specified object, and then removes it from its parent v. If v has two

children after removing α, it updates the keys of nodes on the path from v to the root and the procedure stops. Otherwise, we must merge v that has only one child to one of its siblings. Let v_c be the only child remaining in v. We need to insert v_c as a child to one of v's siblings that is immediately on the left or right hand of v_c and remove the node v from the tree. The key of every resulting node should be updated.

The procedure of DELETE operation is formally described as 2-3-Tree-Delete(T, α) in Algorithm 3. It deletes a subtree rooted at node α from tree T.

Algorithm 3 2-3-Tree-Delete(T, α)

Input: A tree T and a subtree in T rooted at α.

Output: Tree T after removing the subtree rooted at α.

1: Let *root* be the root of T and v be the parent of α.
2: Remove α.
3: If v is the root, update the keys of v and stop.
4: If v has two children, update the keys of nodes on the path from v to *root* and stop.
5: Let u be the only child of v. Let γ be the place of u and disconnect u with v.
6: Call 2-3-Tree-Delete(T, v).
7: Call 2-3-Tree-Insert(T, γ, u).

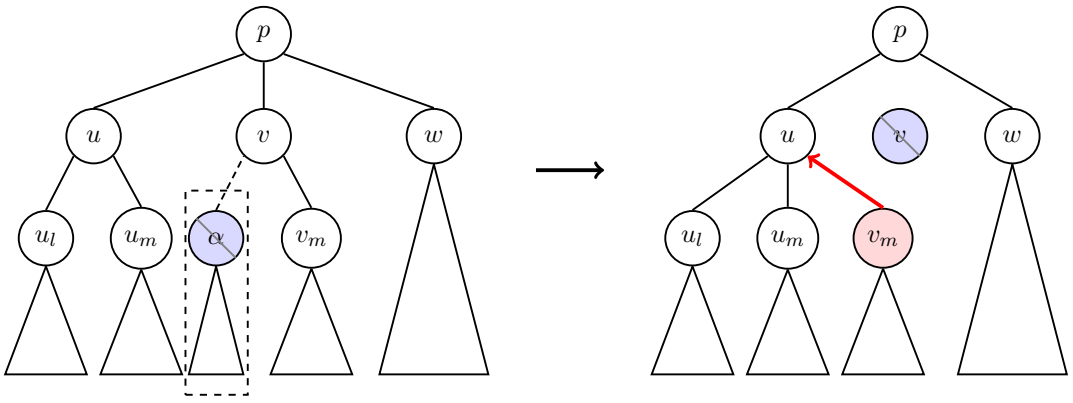


Figure 3.5: An example of deleting a subtree from a 2-3 tree

The processing of deleting a subtree from a 2-3 tree by 2-3-Tree-Delete operation is illustrated through an example in Figure 3.5. We are asked to delete a subtree rooted at node α from the tree. We first remove α from its parent node v. As a result, v becomes a node having only one child v_m. Then, we disconnect v_m from v, reconnect it as a child of

the sibling u of v and delete v from the tree. After we delete v, the parent p of v becomes a node having two children and thus the deletion procedure terminates after the key of every resulting node is updated.

If we want to delete an object with key k from a 2-3 tree rooted at $root$, we can first call procedure 2-3-Tree-Search($root, k$) to find the corresponding leaf α that stores the object and then call procedure 2-3-Tree-Delete(T, α).

The DELETE operation updates the keys and merges the resulting nodes on a path through the tree without going down. Therefore, this operation also requires $O(\log n)$ time.

3.2.3 Heap

Many applications only need to insert objects and get the object having the highest priority. In designing their implementations, special queues, called *priority queues*, are usually very efficient. A priority queue usually supports two basic operations: INSERT that inserts an object into the queue and DELETEMIN that returns the object having the highest priority and removes it from the queue.

There are many data structures that can be used to implement a priority queue. If we use a simple linked list, we can always insert an object at the head of the list in $O(1)$ time and search for the object having the highest priority in $O(n)$ time by traveling through the entire list of size n. Another method is to keep the elements in a sorted linked list. This leads that the cost of INSERT operation becomes $O(n)$ time and that of DELETEMIN becomes $O(1)$ time. Since we cannot call the DELETEMIN operation more times than that we call the INSERT operation, it seems better to implement it with a sorted linked list.

We can also use a binary search tree that we explained in previous sections. Both of the operations INSERT and DELETEMIN require $O(\log n)$ time. However, implementing a priority queue using a binary search may be expensive because a binary search tree supports more operations that are not necessary.

A *heap*, a specialized tree-based data structure, is usually utilized to implement a priority queue. It also requires $O(\log n)$ time for both the INSERT and DELETEMIN operations in the worst case. However, it is proved that the operation INSERT only costs $O(1)$ time on average. The heap is known as one of the most efficient implementations of a priority queue and in fact a priority queue is often referred as a heap.

Heaps can be classified as either a *min heap* or a *max heap*. In a min (resp. max) heap, the objects with smaller (resp. larger) values of keys have higher priority. We only consider the case of min heap. The case of max heap is similar.

We use a binary tree to represent a heap. In a heap, the object having the smallest key

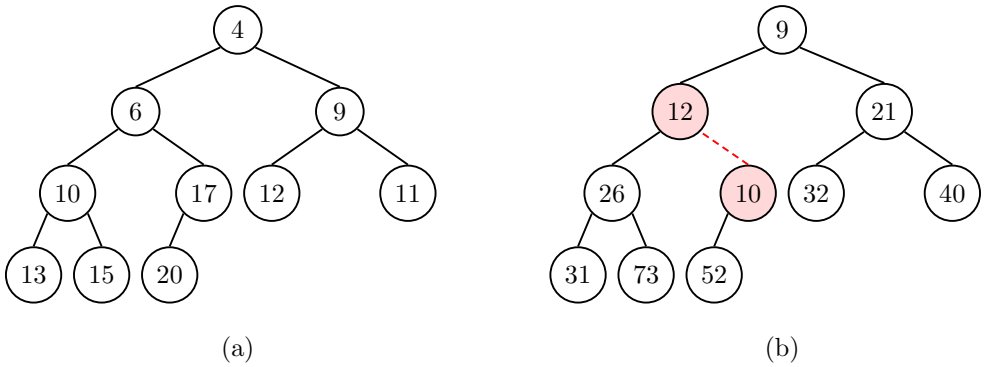


Figure 3.6: An example of heap

is stored at the root and any subtree must also be a heap. This property is called *heap-order property*. Consequently, in the binary tree of a heap, the keys of parent nodes are always smaller than or equal to those of their children and the root contains the smallest value among the keys. Figure 3.6(a) shows an example of a heap, and the binary tree in Figure 3.6(b) is not a heap because the heap-order property is violated between nodes 12 and 10.

It is necessary to maintain the heap-order property when we apply the INSERT and DELETEMIN operations on a heap. The procedures are explained in the following.

Procedure INSERT on a heap

The INSERT operation inserts a new object into a heap without violating heap-order property. We first create an empty leaf node v in the tree at the place of the right side of the rightmost leaf of the deepest level. If the new object with key k can be stored in v without violating heap-order property, i.e., k is not larger than that of v's parent node, we put the new object in v and the procedure terminates. Otherwise, we move the object in the parent node of v to v so that the empty node moves upward to the root. We repeat comparing k with the key of the parent of the empty node and moving the empty node upward until the new object can be put in it. This strategy is called *percolate up* where the empty node percolates up in the heap until finding an appropriate place for the new object.

The procedure of INSERT operation on a heap is formally described as Heap-Insert(T, k) in Algorithm 4. It inserts an object with key k into heap T without violating heap-order property.

Algorithm 4 Heap-Insert(T, k)

Input: A heap T and an object with key k.

Output: Heap T after inserting the object.

1: Create an empty leaf node α on the right side of the rightmost leaf of the deepest level in T. Let u be the parent node of α.

2: If k is not smaller than the key of u, put the object with key k in α and stop.

3: Move the object in u to α and let $\alpha := u$.

4: If α is the root of T, put the object with key k in α and stop. Otherwise, let u be the parent node of α, return to Step 2.

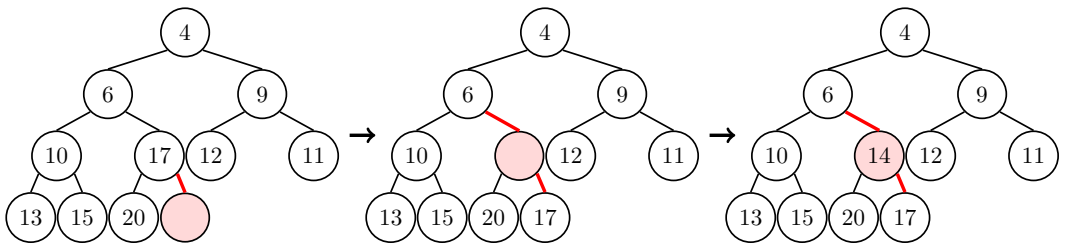


Figure 3.7: An example of inserting an object into a heap

We illustrate the processing of inserting an object with key 14 into a heap with an example in Figure 3.7. We first create an empty leaf node for 14. Because the heap-order property will be violated if we put 14 in the empty node, we move 17 to the empty node. We percolate up the empty node until finding an appropriate place for 14.

Procedure DeleteMin on a heap

The DeleteMin operation can be implemented similarly as the Insert operation on a heap. It finds the object having the smallest key and deletes it from a heap. Finding the object having the smallest key is easy: according to the property of a heap, we can just return the object stored in the root of the heap in $O(1)$ time. The difficult part is to delete the root from a heap.

We remove the object in the root v of a heap and let v be an empty node. We also remove the rightmost leaf node t of the deepest level from the heap and try to put the object in t into v. If the object in t can be stored in v without violating heap-order property, i.e., the key of t is not larger than that of any child of v, we put that object in v and the procedure terminates. Otherwise, we move the object in one of the children of v that has smaller key to v so that the empty node moves downward to the leaf. We

continue moving the empty node until the object in t can be put in it. This strategy is called *percolate down* where we percolate down an empty node in the heap until finding an appropriate place.

The procedure of DeleteMin operation on a heap is formally described as Heap-DeleteMin(T) in Algorithm 5. It returns the object in the root and deletes the object in the root from heap T without violating heap-order property.

Algorithm 5 Heap-DeleteMin(T)

Input: A heap T.

Output: The object in the root of T and T after deleting that object.

1: Let α be the root of T and return the object in α. Let t be the rightmost leaf of the deepest level in T.
2: If α is a leaf, move the object in t to α, and go to Step 6.
3: Let u and v be the children of α and node u has smaller key.
4: If the key of t is not larger than that of u, move the object in t to α, and go to Step 6.
5: Move the object in u to α and let $\alpha := u$. Return to Step 2.
6: Delete t from T and stop.

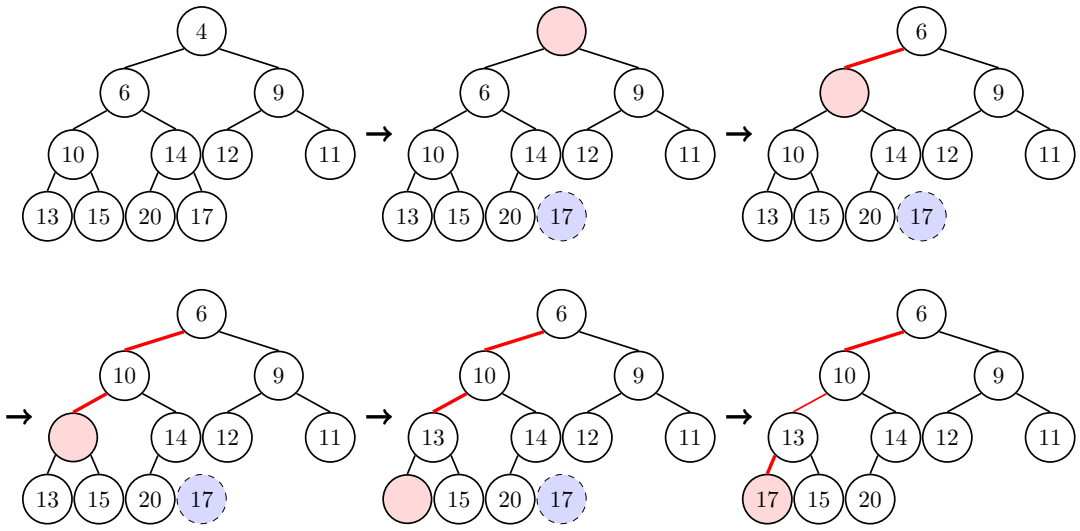


Figure 3.8: An example of deleting the object with the minimum key from a heap

Figure 3.8 shows an example of the processing when we delete the object with the minimum key from a heap. We first remove the object in the root and make it an empty node. Then, we remove node 17 from the heap and try to put it into the empty node. Because the heap-order property will be violated if we put 17 into the empty node at the

root, we move node 6, whose key 6 is smaller than that of the other child node 9, to the empty node. We percolate down the empty node until we find an appropriate place for 17.

3.3 No-fit polygon

The no-fit polygon (NFP) is a geometric technique to check overlaps of two polygons in two-dimensional space. This concept was introduced by Art [7] in 1960s, who used the term "shape envelope" to describe the positions where two polygons can be placed without intersection.

The NFP is defined for an ordered pair of two polygons P_i and P_j, where the position of polygon P_i is fixed and polygon P_j can be moved. The NFP of P_j relative to P_i, $NFP(P_i, P_j)$, denotes the set of all possible positions of polygon P_j having an intersection with polygon P_i. Let $P(x, y)$ be the polygon P placed at (x, y), i.e., the region occupied by P when its reference point is located at (x, y), and $\mathrm{int}(P)$ be the interior of P. We also assume that the position of polygon P_i is fixed at the origin $(0, 0)$. Then we have

$$NFP(P_i, P_j) = \{(x, y) \mid \mathrm{int}(P_i(0, 0)) \cap P_j(x, y) \neq \emptyset\}. \tag{3.3.5}$$

For a point $r = (x, y)$ in the plane and a set $S \subset R^2$, we define $-r = (-x, -y)$ and $-S = \{-r \mid r \in S\}$. In other words, $-S$ is obtained by reflecting S according to the origin. Then, $NFP(P_i, P_j)$ is formally defined by the Minkowski sum

$$NFP(P_i, P_j) = \mathrm{int}(P_i) \oplus (-\mathrm{int}(P_j)) = \{u - w \mid u \in \mathrm{int}(P_i), w \in \mathrm{int}(P_j)\}. \tag{3.3.6}$$

Note that $NFP(P_i, P_j)$ is an open set, i.e., it consists of all points inside of a polygon except for boundary points. Let $\partial NFP(P_i, P_j)$ denote the boundary of $NFP(P_i, P_j)$, and $\mathrm{cl}(NFP(P_i, P_j))$ denote the closure of $NFP(P_i, P_j)$, i.e., $\mathrm{cl}(NFP(P_i, P_j)) = \partial NFP(P_i, P_j) \cup NFP(P_i, P_j)$. The no-fit polygon has the following important properties:

- $P_j(x_j, y_j)$ overlaps with $P_i(x_i, y_i) \iff (x_j, y_j) \in NFP(P_i, P_j) \oplus (x_i, y_i)$.
- $P_j(x_j, y_j)$ touches $P_i(x_i, y_i) \iff (x_j, y_j) \in \partial NFP(P_i, P_j) \oplus (x_i, y_i)$.
- $P_j(x_j, y_j)$ and $P_i(x_i, y_i)$ are separated $\iff (x_j, y_j) \notin \mathrm{cl}(NFP(P_i, P_j)) \oplus (x_i, y_i)$.

When P_i and P_j are both convex, $\partial NFP(P_i, P_j)$ can be computed by the following simple procedure: Slide P_j around P_i having it keep touching with P_i. Then the trace of the reference point of P_j is $\partial NFP(P_i, P_j)$. Such a no-fit polygon $NFP(P_i, P_j)$ is illustrated through the example in Figure 3.9.

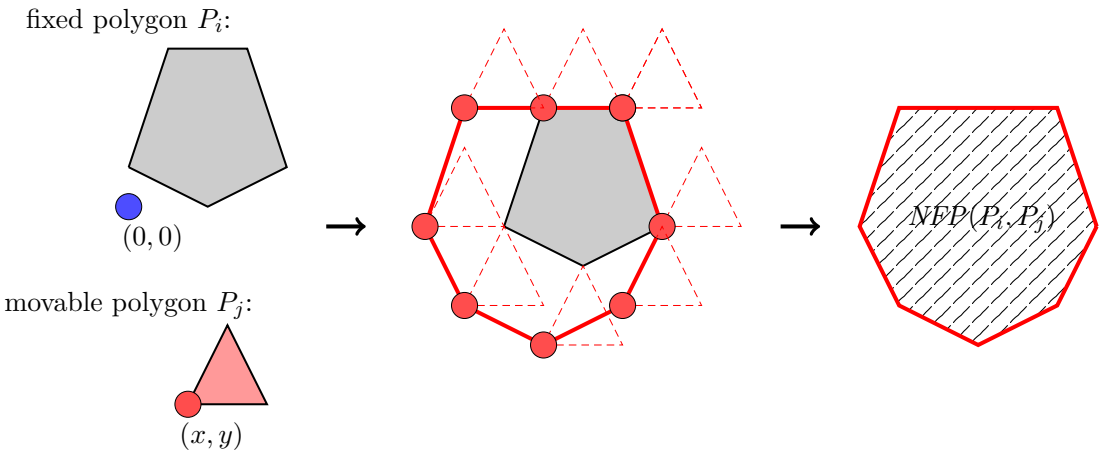


Figure 3.9: An example of $NFP(P_i, P_j)$

Utilizing the technique of no-fit polygon, the problem of checking whether two polygons overlap or not becomes an easier problem of checking whether a point is in a polygon or not. When the two polygons are clear from the context, we may simply use NFP instead of $NFP(P_i, P_j)$.

Through this thesis, we use the technique of no-fit polygon to determine whether two rectilinear blocks overlap each other. We treat each rectilinear block as a set of rectangles whose relative positions are fixed. The NFP of two rectilinear blocks is the union of the no-fit polygons of all pairs of rectangles that represent the two rectilinear blocks. Note that when polygons P_i and P_j are rectangles, $NFP(P_i, P_j)$ is also a rectangle and it can be computed in $O(1)$ time. The method of computing NFP of two rectangles in $O(1)$ time and the details of calculating NFP for rectilinear blocks are explained in Section 4.1.

3.4 Bottom-left stable feasible positions

For a given area where a set of rectilinear blocks are placed and one new item to be placed, a bottom-left stable feasible position (*BL stable feasible positions*) is a location in the area where the new item can be placed without overlap with already placed rectilinear blocks and the new item cannot be moved leftward or downward. In this thesis, we assume that the shape of the given area is rectangular. "Bottom-left stable" means that the new item cannot move to the bottom or to the left, and "feasible" means that the new item will not overlap with other blocks when it is placed.

Note that there are many bottom-left stable feasible positions in general. The *bottom-left position* (*BL position*) is defined as the leftmost location among the lowest bottom-left

stable feasible positions. The bottom-left stable feasible positions and the BL position are illustrated through the example in Figure 3.10.

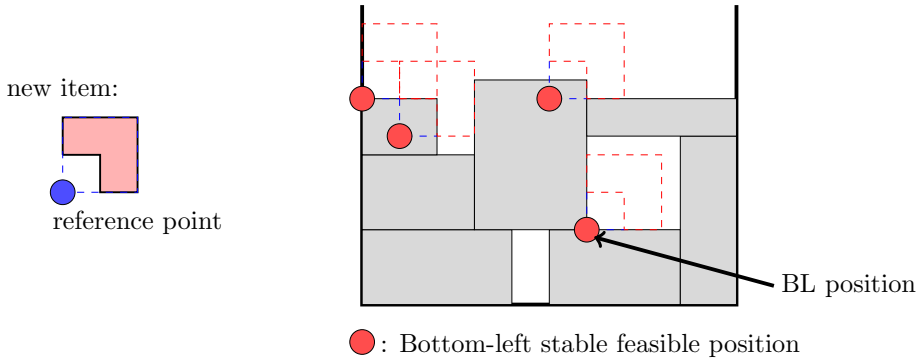


Figure 3.10: Bottom-left stable feasible positions and the BL position

We take the bottom-left strategy as the main strategy of our construction algorithms. Whenever we want to pack a new item into the container, the new item is placed at the BL position relative to the current layout.

3.5 Find2D-BL algorithm

The *Find2D-BL algorithm* [48] was proposed to calculate the BL position for a new rectangle to be placed relative to a rectangular container and to rectangles already placed in the container. The main idea of the Find2D-BL algorithm is to find bottom-left stable feasible positions by using the technique of NFP and a sweep-line method. The *sweep line* is a line parallel to the x-axis, which moves upward from the bottom of the container.

The Find2D-BL algorithm first calculates the NFPs of the new rectangle relative to the rectangles in the container, and it places the NFPs at the positions where the corresponding rectangles are placed. The *overlap number* of a point $v = (x, y)$, denoted by $\pi(x, y)$, is the number of NFPs that contain v. A new rectangle can be placed at a point in the container whose overlap number equals zero. The BL position is the leftmost point in the container among the lowest points whose overlap number is zero. The Find2D-BL algorithm keeps the overlap number of arbitrary points on the sweep line in a complete binary search tree data structure explained in Section 3.2.1. With the sweep line moving upward from the bottom, the bottom-left position appears as the leftmost point among the initially emerging points on the sweep line whose overlap number is zero. It is shown

in [48] that the time complexity of this algorithm is as follows.

Theorem 3.5.1. *(Imahori et al. [48]) Assuming that the number of rectangles is δ, the Find2D-BL algorithm finds the BL position for the new rectangle in $O(\delta \log \delta)$ time.*

3.6 Heuristics for the rectangle packing problem

In this section, we explain two representative construction heuristics for the rectangle packing problem, the *bottom-left* and the *best-fit* algorithms. We first explain the bottom-left algorithm, which is one of the simplest forms among the algorithms based on the bottom-left strategy. Then we explain the best-fit algorithm, which is slightly more complicated than the bottom-left algorithm but it is known to be more effective.

3.6.1 Bottom-left algorithm for the rectangle packing problem

The *bottom-left algorithm* for the rectangle packing problem was proposed by Baker et al. [9]. The algorithm uses the bottom-left strategy to pack items one by one at their BL positions into the strip according to a given sequence. They also showed that when the sequence of rectangles is the decreasing order of their widths, the resulting height of the container will not be more than three times that of an optimal packing layout.

The bottom-left algorithm is one of the simplest forms among the algorithms based on the bottom-left strategy and it performs fairly well in practice. Hence, the implementations for the bottom-left algorithm have been studied for decades. If naively implemented, the bottom-left algorithm requires $O(\delta^4)$ time, where δ is the number of given rectangles. Some simple implementations for this algorithm that require $O(\delta^3)$ time in the worst case are also developed. One typical implementation with running time of $O(\delta^3)$ was proposed by Hopper and Turton in [43]. Chazelle [20] proposed an $O(\delta^2)$ time and $O(\delta)$ space implementation for the bottom-left algorithm.

3.6.2 Best-fit algorithm for the rectangle packing problem

Burke et al. [19] proposed a different type heuristic algorithm based on the bottom-left strategy, called the *best-fit algorithm*. The best-fit heuristic algorithm is also a greedy algorithm. It is slightly more complicated than the bottom-left algorithm but it is known to be more effective.

Instead of using a sequence of rectangle in the bottom-left algorithm, the best-fit algorithm dynamically selects the next rectangle to pack during the packing process. It always examines an available space as low as possible in the strip and then places the rectangle that best fits the space.

The best-fit algorithm also packs rectangles one by one at their BL positions. During the packing process, the algorithm maintains a *skyline* of the upper edges of all placed rectangles in the container. The skyline of a packing layout is the set of points that can be seen from an infinitely high position. A skyline consists of a set of *segments* where each segment touches the upper edge of at least one rectangle in the container or the bottom edge of the strip. Any two adjacent segments have different heights and exactly one common x-coordinate. Figure 3.11 shows an example of the skyline of a packing layout. Each dashed line represents a segment and the union of all the segments is the skyline of the packing layout in Figure 3.11.

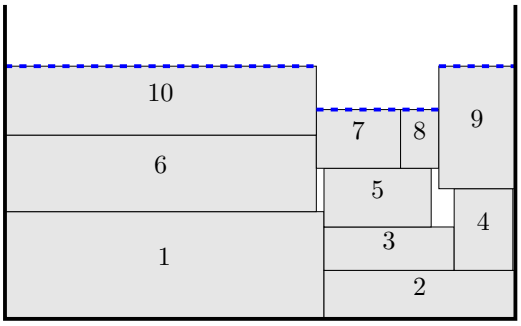


Figure 3.11: An example of the skyline of a packing layout

For each iteration, the best-fit algorithm first finds the *lowest available segment*, the segment having the lowest height (the smallest y-coordinate) in the skyline. If there exist more than one segment having the lowest height, the algorithm chooses the leftmost one among them as the lowest available segment. If there are rectangles that have not been placed yet and can be placed on the segment (i.e., the width of such a rectangle is not larger than that of the lowest available segment), the best-fit algorithm selects the widest rectangle (the rectangle with the largest width) and places it at the leftmost position on the segment. Whenever a rectangle is placed, the algorithm updates the skyline relative to the resulting packing layout. If there are no rectangles that can be placed on the lowest available segment, the algorithm raises the segment to the height of its lower adjacent segment and merge the two segments (if both its adjacent segments have the same height, three of these segments should be merged). The best-fit algorithm repeats this procedure until all the rectangles are placed into the container.

Assume that δ is the number of given rectangles. If naively implemented, the best-fit algorithm requires $O(\delta^4)$ time in the worst case. Imahori and Yagiura [49] proposed an efficient implementation of the best-fit algorithm that requires linear space and $O(\delta \log \delta)$

time, where δ is the number of rectangles.

Chapter 4

Construction Heuristics for the Rectilinear Block Packing Problem

In this chapter, we explain the bottom-left and best-fit algorithms for the rectilinear block packing problem, which are generalized from the algorithms for the rectangle packing problem. A crucial problem in generalizing these algorithms is how to find the bottom-left position of a new rectilinear block. We first introduce a method to calculate NFPs for rectilinear blocks in Section 4.1. Then in Section 4.2, we explain an algorithm that we call *Find2D-BL-R* to find the BL position of a rectilinear block. We explain how to generalize the bottom-left and best-fit algorithms to solve the rectilinear block packing problem in Section 4.3 and 4.4, and analyze their time complexities. Finally, we also explain in Section 4.5 the time complexities of the two algorithms when naively implemented.

4.1 Method of calculating NFPs for rectilinear blocks

In our algorithms, we assume that each rectilinear block R_i is represented with a set $\mathcal{B}_i$ of rectangles whose relative positions are fixed, and let m_i be the size of $\mathcal{B}_i$, (i.e., m_i be the number of rectangles that represents a rectilinear block R_i). We also assume that each such rectangle has a positive area, i.e., special cases of rectangles such as line segments and points are not considered. Note that, a rectilinear block with $\tilde{m}_i$ *concave* vertices (i.e., vertices whose angle outside of the block is 90°) can be cut into at most $\tilde{m}_i+1$ rectangular pieces by horizontal lines that go through its concave vertices. Hence, a rectilinear block with $\tilde{m}_i$ concave vertices can be represented by at most $\tilde{m}_i+1$ rectangles, i.e., $m_i \leq \tilde{m}_i+1$. It is also noted that there is no restriction on the way the relative positions are fixed, e.g.,

an item R_i can be a set of two separate rectangles as long as their relative positions are fixed. Hence, our algorithms can also deal with the packing problem in which each item can be a set of (non-overlapping) rectilinear blocks whose relative positions are fixed.

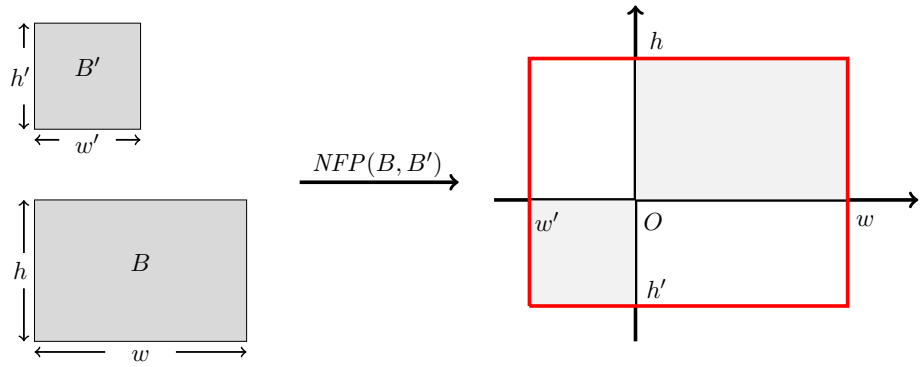


Figure 4.1: NFP of two rectangles

When we are given two rectangles B and B', where rectangle B (resp., B') has width w (resp., w') and height h (resp., h'), $NFP(B, B')$ can be computed by the following expression:

$$NFP(B, B') = \{(x, y) \mid -w' < x < w,\ -h' < y < h\}. \tag{4.1.1}$$

Note that $NFP(B, B')$ is a rectangle, and it can be computed in $O(1)$ time. Figure 4.1 shows an example of $NFP(B, B')$.

We then consider the case when two rectilinear blocks R_i and R_j are given. Let $\mathcal{B}_i = \{B_{i1}, B_{i2}, \ldots, B_{i,m_i}\}$ be the set of rectangles that represents R_i, and v_{ik} be the position of B_{ik} relative to the reference point of R_i for $k = 1, 2, \ldots, m_i$, i.e.,

$$R_i = \bigcup_{k=1}^{m_i} (B_{ik} \oplus v_{ik}). \tag{4.1.2}$$

The set $\mathcal{B}_j$ is defined similarly. The $NFP(R_i, R_j)$ is the union of $NFP(B_{ik}, B_{jl})$ for all pairs of B_{ik} and B_{jl}. That is, $NFP(R_i, R_j)$ can be formally calculated as follows:

$$NFP(R_i, R_j) = \bigcup_{k=1}^{m_i} \bigcup_{l=1}^{m_j} (NFP(B_{ik}, B_{jl}) \oplus v_{ik} \oplus (-v_{jl})). \tag{4.1.3}$$

For each rectangle B_{jl}, we can easily calculate its NFP with respect to B_{ik} by using (4.1.1). Hence $NFP(R_i, R_j)$ consists of $m_i m_j$ rectangles, and it can be computed in $O(m_i m_j)$ time. For convenience, we call such rectangles *NFP rectangles*.

For the case of rectilinear block packing, we define the overlap number $\pi(x, y)$ of a point (x, y) to be the number of NFP rectangles containing it. Let $\mathcal{N}$ denotes the set of all NFP rectangles in the plane and assume that every NFP rectangle in $\mathcal{N}$ is placed at the position where the corresponding item is placed. Then the overlap number $\pi(x, y)$ is formally described as follows:

$$\pi(x, y) = |\{N_k \mid (x, y) \in N_k,\ N_k \in \mathcal{N}\}|\,. \tag{4.1.4}$$

For a set $\tilde{\mathcal{R}} \subseteq \mathcal{R}$ of placed items and a rectilinear block R_j to be placed, where each item $R_i \in \tilde{\mathcal{R}}$ is placed at v_i, $\mathcal{N}$ consists of all NFP rectangles in $NFP(R_i, R_j) \oplus v_i$ for all rectilinear block R_i placed. Then from (4.1.3), R_j can be placed at a point (x, y) without overlap with any item in $\tilde{\mathcal{R}}$ if and only if $\pi(x, y) = 0$.

4.2 Method of calculating BL position for rectilinear block

In this section, we explain the Find2D-BL-R algorithm, which uses the Find2D-BL algorithm, to find the BL position for a rectilinear block.

First we calculate NFPs of the new rectilinear block relative to all of the items in the container and then place each of them at the position where the corresponding item is placed. Next, all we have to do to compute the BL position is to find the leftmost point in the container among the lowest positions whose overlap number is zero. According to the method of calculating NFPs explained in Section 4.1, the NFPs of the new rectilinear block to be placed relative to items in the container consist of rectangles. The Find2D-BL algorithm explained in Section 3.5 can find the BL position for a new rectangle relative to rectangles already placed in the container. Hence, we can use the Find2D-BL algorithm to the case of rectilinear blocks. After we obtain the NFPs of the new rectilinear block that consist of rectangles, we can find the BL position by calling the Find2D-BL algorithm.

The Find2D-BL-R algorithm is formally described as Find2D-BL-R(R_j, L) in Algorithm 6 where R_j is the new rectilinear block to be packed and L is the current packing layout.

Algorithm 6 Find2D-BL-R(R_j, L)

Input: A block R_j and current packing layout L in the container.

Output: The BL position of R_j.

1: Calculate NFPs of R_j relative to the current packing layout L by using expression 4.1.3.
2: Place the NFPs at the positions where the corresponding items are placed.
3: Call Find2D-BL algorithm to compute the BL position (x_j, y_j) of R_j.
4: Output point (x_j, y_j) and stop.

The number of rectangles that represent all items R_i in the container is not more than M where $M = \sum_{i=1}^{n} m_i$. Hence, the number of rectangles that constitute the NFPs of R_j relative to the packing layout is not more than $m_j M$. The running time for calculating NFPs of R_j in Step 1 and for placing them in Step 2 is $O(m_j M)$. According to Theorem 3.5.1, the Find2D-BL algorithm finds the BL position in $O(\delta \log \delta)$ time when the number of placed rectangles is δ. Since the number of placed rectangles is $O(m_j M)$, the running time of Step 2 is $O(m_j M \log(m_j M)) = O(m_j M \log M)$ time. Therefore, the Find2D-BL-R algorithm computes the BL position of a rectilinear block R_j in $O(m_j M \log M + m_j M) = O(m_j M \log M)$ time.

Theorem 4.2.1. *Assuming that M is the number of rectangles that represent all items R_i in the container and m_j is the number of rectangles that represent a rectilinear block R_j, the Find2D-BL-R algorithm computes the BL position of R_j in $O(m_j M \log M)$ time.*

4.3 Bottom-left algorithm for the rectilinear block packing problem

In this section, we generalize the bottom-left algorithm for the rectangle packing problem explained in Section 3.6.1 to solve the rectilinear block packing problem.

The bottom-left algorithm can be generally explained as follows: Given a set of n rectilinear blocks $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and an order of items (e.g., decreasing order of area), the algorithm packs all of the items one by one according to the given order, where each item is placed at its BL position relative to the current layout (i.e., the layout at the time just before it is placed).

Assume for simplicity that $\{R_1, R_2, \ldots, R_n\}$ are packed according to the increasing order of their indices. The bottom-left algorithm, in which Find2D-BL-R is utilized to find BL positions, is formally described in Algorithm 7.

Algorithm 7 Bottom-Left_Find2D-BL-R

Input: A sequence of rectilinear blocks $R_1, R_2, \ldots, R_n$ and a container.

Output: A packing layout.

1: Set $j := 0$ and the current packing layout $L := \emptyset$.
2: Set $j := j + 1$. If $j > n$, output L and stop.
3: Call Find2D-BL-R(R_j, L) to find the BL position (x, y) of R_j in L.
4: Pack R_j at (x, y) in L, and then return to Step 2.

Recall that the number of rectangles that represents a rectilinear block R_i is denoted by m_i, and M denotes the sum of m_i over all the n rectilinear blocks. According to

Theorem 4.2.1, the Find2D-BL-R algorithm calculates the BL position of a new rectilinear block R_j in Step 3 in $O(m_j M \log M)$ time. The computation of other steps is less expensive than this. The bottom-left algorithm therefore runs in $\sum_{j=1}^{n} O(m_j M \log M) = O(M^2 \log M)$ time.

Theorem 4.3.1. *Assuming that M is the number of rectangles that represent all rectilinear blocks, the bottom-left algorithm runs in $O(M^2 \log M)$ time.*

4.4 Best-fit algorithm for the rectilinear block packing problem

In this section, we generalize the best-fit algorithm for the rectangle packing problem explained in Section 3.6.1 to solve the rectilinear block packing problem. It seems hard to directly generalize this original idea of the best-fit for the rectangle packing problem explained in Section 3.6.2 to the case of rectilinear blocks. We first give an interpretation of the best-fit algorithm and this will make the generalization possible.

The following is a different explanation of the best-fit algorithm. To be more precise, for any instance of the rectangle packing problem, the solution obtained by the following procedure is exactly the same as the one obtained by that introduced in Section 3.6.2. At the beginning of the packing process, no rectangles are placed in the container. The algorithm packs rectangles one by one at their BL positions, and in each iteration, it dynamically selects a rectangle to pack among the remaining items by the following rule: Calculate the BL positions of all of the remaining items. Then, in this iteration, the rectangle whose BL position takes the smallest x-coordinate among those with the lowest y-coordinate is packed. When there is more than one such rectangle, the widest one (resolving equal widths by the largest height) is chosen.

The best-fit algorithm for the rectilinear block packing problem is explained as follows: Given a set of n rectilinear blocks $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and a priority among them (e.g., an item with a wider bounding box has higher priority), the algorithm packs all of the items one by one into the container, where each item is placed at its BL position relative to the current layout. At the beginning of the packing process, no item is placed in the container. Whenever an item is to be packed into the container, the algorithm calculates the BL positions of all of the remaining items relative to the current layout. In this iteration, the rectilinear block whose BL position takes the smallest x-coordinate among those with the lowest y-coordinate is packed. If there exist ties, the block with the highest priority is chosen.

In the above algorithm, "priority" generalizes the idea of choosing wider rectangles in

the best-fit algorithm for rectangles. That is, when the given items are all rectangles and the priority is defined such that wider items have higher priority (resolving equal widths by the largest height), the above algorithm becomes the same as the best-fit algorithm for rectangles. For rectilinear blocks, however, such an interpretation seems impossible, and various rules for the priority can be considered. In the computational experiments in Section 5.6, we test some basic rules for deciding the priority.

The best-fit algorithm for the rectilinear block packing problem can be implemented by using the Find2D-BL-R algorithm as follows. In each iteration, after a rectilinear block is packed into the container, the y-coordinate of its reference point is recorded as *currentBottom*. Note that, there are no rectilinear blocks that can be placed below the line whose y-coordinate is *currentBottom*. We can therefore discard the space below the line $y = currentBottom$ from the candidates for the BL position. For this reason, the Find2D-BL-R algorithm can set the initial position of the sweep line to $y = currentBottom$, instead of $y = -\infty$ ("$y = -\infty$" signifies a sufficiently low position where the sweep line overlaps with no NFP). This significantly reduces the computation time in practice, although the worst-case time complexity stays the same as in the case where the sweep line always starts from $y = -\infty$. The best-fit algorithm is formally described in Algorithm 8.

Algorithm 8 Best-Fit_Find2D-BL-R

Input: A set of rectilinear blocks $\mathcal{R}$ and a container.

Output: A packing layout.

1: Set $\mathcal{R}' := \mathcal{R}$, the current packing layout $L := \emptyset$ and $currentBottom := -\infty$.
2: If $\mathcal{R}' = \emptyset$, output L and stop.
3: Set $\mathcal{R}'' := \mathcal{R}'$, $x^* := +\infty$ and $y^* := +\infty$.
4: Choose an item R_j in $\mathcal{R}''$, and then let $\mathcal{R}'' := \mathcal{R}'' \setminus \{R_j\}$.
5: Call Find2D-BL-R(R_j, L) to find the BL position (x, y) of R_j, in which the initial position of the sweep line is set to *currentBottom*.
6: If one of the following three conditions holds, then let $x^* := x$, $y^* := y$ and $j^* := j$.

(i) $y < y^*$.

(ii) $y = y^*$ and $x < x^*$.

(iii) $y = y^*$, $x = x^*$ and R_j has higher priority than R_{j^*}.

7: If $\mathcal{R}'' \neq \emptyset$, return to to Step 4.
8: Pack the item R_{j^*} so that its reference point is placed at (x^*, y^*) in L.
9: Set $\mathcal{R}' := \mathcal{R}' \setminus \{R_{j^*}\}$ and $currentBottom := y^*$. Return to Step 2.

Recall that each rectilinear block takes a deterministic shape from the set of t shapes $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$, and when $t < n$, some items have an identical shape. In Step 6, if the bottom-left position of an item with the same shape as R_j has already been computed after Step 3 was executed most recently, it is not necessary to compute the bottom-left position of R_j, because it is the same as that of the item with the same shape as R_j. Hence, we can reduce the number of calls to Find2D-BL-R by not invoking it when the bottom-left position of an item with the same shape has already been computed. Accordingly, whenever the algorithm packs a new rectilinear block into the container, the Find2D-BL-R algorithm is called at most once for each shape.

According to Theorem 4.2.1, the Find2D-BL-R algorithm calculates the BL position of a rectilinear block R_j in Step 6 in $O(m_j M \log M)$ time. In the loop from Step 4 to 7 until it exits to Step 8, the computation of BL positions is executed at most once for every shape in $\mathcal{T}$. The computation time of this loop is therefore $O(mM \log M)$, where $m = \sum_{i=1}^{t} m_i^{\mathrm{T}}$ for m_i^{T} defined to be the number of rectangles that represent T_i, i.e., $m_i^{\mathrm{T}} = m_j$ for an item R_j whose size and shape is T_i. This loop is executed whenever an item is to be placed, and the number of times an item is placed is n. Therefore, the above implementation of the best-fit algorithm runs in $O(nmM \log M)$ time.

Theorem 4.4.1. *Assuming that M is the number of rectangles that represent all n rectilinear blocks and m is that of rectangles represent all distinct shapes of rectilinear blocks, the best-fit algorithm packs n rectilinear blocks in $O(nmM \log M)$ time.*

4.5 Time complexities of heuristic algorithms with naive implementations

For comparison purposes, we explain the time complexities of the bottom-left and the best-fit algorithms when they are naively implemented.

In the naive implementation, we also utilize the technique of NFP to calculate overlap numbers of rectilinear blocks. Observe that the BL position of a rectilinear block only appear at a crossing point where an NFP rectangle's right edge crosses another's top edge. Such a crossing point is denoted as *CrossPoint*. Assuming that the number of NFP rectangles is δ, the computation time to find the BL position is $O(\delta^3)$, because there are $O(\delta^2)$ CrossPoints and it takes $O(\delta)$ time to calculate the overlap number for each CrossPoint. The number of rectangles that represent a rectilinear block R_i is denoted by m_i and let $M = \sum_{i=1}^{n} m_i$. We also define m_j^{T} as the number of rectangles that represent shape T_j, and m is the sum of m_j^{T} for all of the shapes in $\mathcal{T}$. The bottom-left and the best-fit algorithms compute the BL position of a rectilinear block R_i in $O((m_i M)^3)$ time,

because the number of NFP rectangles is the sum of $m_i m_j$ for all items R_j in the container, which is $O(m_i M)$.

For every iteration, when the bottom-left algorithm packs a rectilinear block R_i into the container, it calculates the NFPs in $O(m_i M)$ time and computes the BL position of R_i in $O((m_i M)^3)$ time. The bottom-left algorithm therefore runs in $\sum_{i=1}^{n} O((m_i M)^3)$ time. Noting that $\sum_{i=1}^{n}(m_i M)^3 \leq \sum_{i=1}^{n}(mM)^3 = nm^3M^3$ and $\sum_{i=1}^{n}(m_i M)^3 \leq (\sum_{i=1}^{n} m_i M)^3 = M^6$, the time complexity of the bottom-left algorithm is $O(\min\{M^6, nm^3M^3\})$.

For every iteration, when the best-fit algorithm packs a rectilinear block into the container, it computes the BL positions for all the shapes, and the running time for calculating NFPs is $\sum_{j=1}^{t} O(m_j^{\mathrm{T}} M) = O(mM)$ and for computing the BL positions is $\sum_{j=1}^{t} O((m_j^{\mathrm{T}} M)^3) = O(m^3M^3)$. The best-fit algorithm therefore runs in $\sum_{i=1}^{n} O(m^3M^3) = O(nm^3M^3)$ time.

4.6 Conclusion

In this chapter, we generalized two well-known construction heuristics for the rectangle packing problem, the bottom-left and the best-fit algorithms, to solve the rectilinear block packing problem. We also gave an efficient implementations for these two algorithms. If naively implemented, the bottom-left algorithm requires $O(\min\{M^6, nm^3M^3\})$ time and the best-fit algorithm requires $O(nm^3M^3)$ time. We generalized the algorithm proposed in [48] to find the BL position efficiently and reduced the running time of the bottom-left algorithm to $O(M^2 \log M)$ and that of the best-fit algorithm to $O(nmM \log M)$.

Although we performed a series of experiments based on benchmark instances, we did not report any computational results in this chapter. This is because we design more efficient implementations for these two algorithms in the next chapter. For comparison purposes, we report the computational results in the next chapter.

Chapter 5

Efficient Implementations of Construction Heuristics

In this chapter, we design more efficient implementations of the bottom-left and best-fit algorithms than those explained in Chapter 4.

The basic idea of the efficient implementations is introduced in Section 5.1, and the details for calculating BL positions using sophisticated data structures are explained in Section 5.2. In Section 5.4, we explain how the data structures in Section 5.2 are utilized to make the bottom-left and best-fit algorithms faster and then analyze the time complexity of these algorithms.

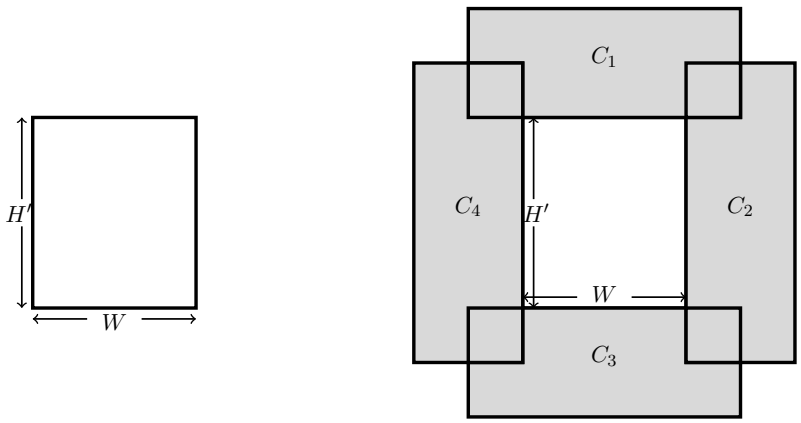


Figure 5.1: Container Rectangles

Instead of considering the constraint that we must place every rectilinear block inside of the container, we use a set of four sufficiently large rectangles called *container rectangles*

$\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ that satisfies the following condition: Assuming that $\tilde{\mathcal{R}}$ is the set of items that are arbitrarily placed in the container, a rectilinear block R_i is placed in the container without protrusion and without overlap with any item in $\tilde{\mathcal{R}}$ if and only if R_i does not have overlap with any item in $\tilde{\mathcal{R}} \cup \mathcal{C}$. We denote $\hat{\mathcal{R}} = \tilde{\mathcal{R}} \cup \mathcal{C}$; then $|\hat{\mathcal{R}}| = |\tilde{\mathcal{R}}| + 4$ holds. See Figure 5.1 for an example of container rectangles, where the value of H' is $+\infty$. Figure 5.1 (a) shows the give area, i.e., the container, and Figure 5.1 (b) shows the corresponding container rectangles.

5.1 Basic idea

In this section, we explain the basic idea to compute BL positions efficiently for construction heuristics that are based on the bottom-left strategy.

Recall that when the Find2D-BL-R algorithm computes the BL position of an item R_j, it uses the NFPs of R_j relative to the items in the container, and such NFPs are placed at the positions where the corresponding items are placed. We call such a layout of NFPs an *NFP layout* for R_j. If the shapes of two items R_j and $R_{j'}$ are the same, their NFP layouts are the same; thus it suffices to have $t = |\mathcal{T}|$ NFP layouts, each for a distinct shape in $\mathcal{T}$, to compute the BL positions of all remaining items.

The basic idea is to dynamically keep the NFP layouts with respect to the current packing layout for all shapes in $\mathcal{T}$ during the packing process. In other words, we do not compute NFP layouts from scratch in each iteration of the construction heuristics.

Because the algorithm needs to keep t NFP layouts (and related data structures for each NFP layout), $O(t)$ times more memory space is necessary compared to the implementations in Section 4.3 and 4.4 in which the Find2D-BL-R algorithm is invoked whenever needed. A common feature of construction heuristics is that once an item is packed into the container, its position is fixed and will not change. This means that for each shape T_j, after packing an item R_i into the container, the NFPs in the container do not change, and we can obtain the new NFP layout for T_j simply by inserting $NFP(R_i, T_j)$. Thus, for each shape, we dynamically modify the NFP layout with respect to the current packing layout during the packing process. Figure 5.2 is an example that shows how NFP layout changes after packing a rectilinear block into the container. Figure 5.2 (a) and (b) are the packing layout and the NFP layout before packing item R_i, and Figure 5.2 (c) and (d) are the resulting packing layout and NFP layout after packing R_i.

Whenever an item R_i is placed into the container, the algorithm computes the BL position of every shape T_j using the NFP layout for T_j and a sweep line parallel to the x-axis, which moves upward and keeps the overlap number $\pi(x, y)$ of every point (x, y) on the sweep line. As discussed in Section 3.5, the BL position is found when a point (x, y)

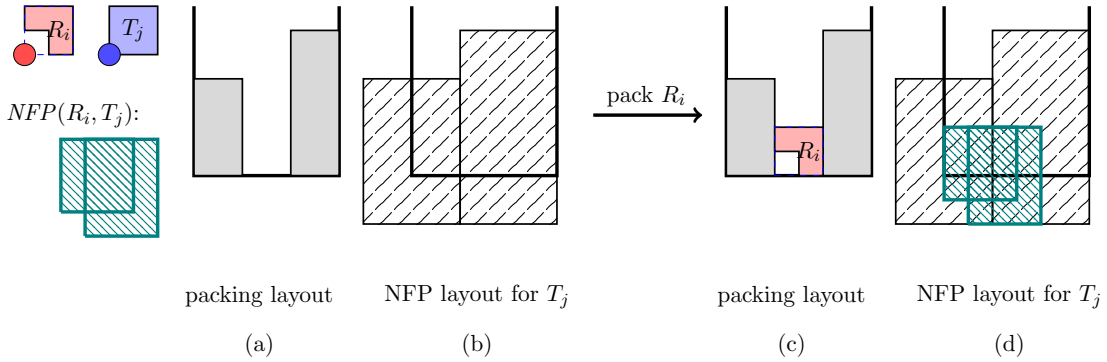


Figure 5.2: An example of NFP layout after packing an item

with $\pi(x, y) = 0$ is found on the sweep line for the first time. Note that the y-coordinate of the BL position of shape T_j does not decrease when item R_i is placed. Hence the BL position of T_j does not appear below the sweep line at the position where it stopped when the last BL position was found before the item R_i was placed. In other words, the sweep line can start the search for the BL position from the last height y_{last} when the last BL position was found before R_i was packed into the container. This means that for each shape T_j, the sweep line moves from bottom to top only once (except for the computation necessary to update the overlap numbers $\pi(x, y_{\text{last}})$ on the sweep line at the last height after $NFP(R_i, T_j)$ is placed) during the entire process of a construction heuristic.

We design sophisticated data structures to dynamically keep the information we need to maintain the overlap numbers of all points on the sweep line. When the sweep line moves from bottom to top, the overlap numbers on the sweep line change only at the positions where the sweep line meets the bottom or top edge of an NFP rectangle. Such a position (or the occasion when the sweep line meets such a position) is called an *event*. For each shape T_j, we maintain a heap $HEAP_j$ for events of the sweep line and a balanced search tree $TREE_j$ to keep the information of overlap numbers on the sweep line. For the balanced search tree, we use a 2-3 tree data structure explained in Section 3.2.2 to implement our algorithm.

For every iteration, after an item R_i is placed, the NFP of shape T_j relative to R_i, which consists of $m_i m_j^{\mathrm{T}}$ NFP rectangles, is added to the NFP layout of T_j for each $j = 1, \ldots, t$. In the following sections, we show that $TREE_j$ and $HEAP_j$ can be updated in $O(\log M)$ time for each j whenever an NFP rectangle is added to the NFP layout of T_j. This implies that after an item R_i is placed, the computation time to update the balanced search trees and heaps for all shapes is $\sum_{j=1}^{t} O(m_i m_j^{\mathrm{T}} \log M) = O(m_i m \log M)$, where m is the sum of m_j^{T} for all the shapes in $\mathcal{T}$ and M is the sum of m_i for all rectilinear blocks in $\mathcal{R}$.

We also show that using $TREE_j$, it is possible in $O(\log M)$ time to judge whether the BL position exists on the current sweep line, and to output such a position if it does exist, whenever the sweep line moves from one event point to another. The number of such events taken from $HEAP_j$ is shown to be $O(\sum_{i=1}^{n} m_i m_j^T) = O(m_j^T M)$ for each shape T_j during the entire computation of a construction heuristic, where $m_j^{\mathrm{T}} M$ is the total number of NFP rectangles in the NFP layout of T_j when the construction algorithm terminates. This implies that the total computation time to search for BL positions is $O(m_j^T M \log M)$ for each shape T_j during the entire execution of the construction heuristic. Because for construction heuristics considered in this thesis, it suffices to update these data structures for all items R_i in $\mathcal{R}$ and to compute BL positions for all shapes T_j in $\mathcal{T}$, the total computation time to maintain these data structures is $O(\sum_{i=1}^{n} m_i m \log M + \sum_{j=1}^{t} m_j^T M \log M) = O(mM \log M)$ during the entire computation of the construction heuristics.

5.2 Method of calculating BL positions

In this section, we explain the details of the efficient method to calculate the BL position of a rectilinear block. In Section 5.2.1, we give a technique to compute the overlap number for each point on the sweep line using a 2-3 tree. The algorithm of calculating a BL position is explained in Section 5.2.2.

5.2.1 Compute overlap numbers by a 2-3 tree

Instead of the binary search tree that are used in the implementation explained in Chapter 4 we use a 2-3 tree to keep the overlap number on the sweep line.

Given a rectilinear shape T_j, to compute the overlap number of each point on the sweep line relative to the current layout, the algorithm first calculates the no-fit polygon $NFP(R, T_j)$ of T_j relative to the placed items $R \in \hat{\mathcal{R}}$ including container rectangles and then places each of them at the position where the corresponding item is placed. (Recall that $\hat{\mathcal{R}}$ is the set of placed items in the container and container rectangles.) Then we have the NFP layout of T_j with respect to the current packing layout. Note that the algorithm actually places the NFP rectangles rather than the NFPs of rectilinear shapes.

Let N_{t} (resp., N_{b}) be the set of all the top (resp., bottom) edges of NFP rectangles that constitute the NFP layout of T_j relative to the items in $\hat{\mathcal{R}}$, and let $N_{\mathrm{tb}} = N_{\mathrm{t}} \cup N_{\mathrm{b}}$. The overlap numbers of points on the sweep line changes only when the sweep line encounters an element in N_{tb}, and it occurs only for the points between the left edge and right edge of the no-fit polygon encountered by the sweep line.

Let $N_{\rm l}$ (resp., $N_{\rm r}$) be the set of all the left (resp., right) edges of NFP rectangles that constitute the NFP layout of T_j, and let $N_{\rm lr} = N_{\rm l} \cup N_{\rm r}$. Note that, the no-fit polygons of T_j relative to the container rectangles in $\mathcal{C}$ can be simply calculated by treating item T_j as a rectangle whose height and width are the same as its bounding box. Hence, for each $i = 1, \ldots, 4$, we use the NFP of the bounding box of T_j relative to container rectangle C_i instead of $NFP(C_i, T_j)$. However, we may not clearly mention this point and assume for simplicity that the four no-fit polygons of T_j relative to the container rectangles consist of four rectangles throughout the remainder of this paper.

Recall that $\tilde{\mathcal{R}}$ is the set of items placed in the container, and let $\tilde{M} = \sum_{R_i \in \tilde{\mathcal{R}}} m_i$. Then there are $m_j^{\rm T}\tilde{M}$ NFP rectangles in the NFP layout of T_j. Hence, $|N_{\rm t}| = |N_{\rm b}| = |N_{\rm l}| = |N_{\rm r}| = m_j^{\rm T}\tilde{M} + 4$ and $|N_{\rm tb}| = |N_{\rm lr}| = 2m_j^{\rm T}\tilde{M} + 8$ hold. The elements in $N_{\rm lr}$ are sorted in nondecreasing order of their x-coordinates, where ties are broken by treating the elements in $N_{\rm r}$ as having higher priority. This tie-breaking rule is important, because if two no-fit polygons have their left and right boundaries at the same x-coordinate, the intersection point of the boundaries of the two no-fit polygons might be a feasible point where the shape T_j can be placed without overlap. Let $x_{\rm lr}^{(k)}$ be the x-coordinate of the kth element in the sorted list of $N_{\rm lr}$, and define intervals

$$S_k = [x_{\rm lr}^{(k)}, x_{\rm lr}^{(k+1)}],\ k = 1, 2, \ldots, 2m_j^{\rm T}\tilde{M} + 7$$

on the sweep line. The left boundary of S_1 (resp., the right boundary of $S_{2m_j^{\rm T}\tilde{M}+7}$) corresponds to the left (resp., right) edge of the container rectangle whose right (resp., left) edge represents the left (resp., right) boundary of the container. The algorithm maintains the overlap number for each interval S_k during the process, where the overlap number $\pi(x, y)$ of a point (x, y) is the number of NFP rectangles containing (x, y) that constitute the NFP layout of T_j. Initially, the sweep line is at a sufficiently low position, and it overlaps with no NFP rectangle. At this moment, the overlap number of every interval S_k is zero. Figure 5.3 (a) shows an example of the intervals corresponding to an NFP layout that contains three NFP rectangles.

We use a 2-3 tree whose leaves represent intervals $S_1, S_2, \ldots, S_{2m_j^{\rm T}\tilde{M}+7}$, where the kth leaf (called leaf k for simplicity) from the left corresponds to the interval S_k. Note that the height of the 2-3 tree is $O(\log(m_j^{\rm T}\tilde{M})) = O(\log M)$. Every node of this tree stores values $p_{\rm value}$, $p_{\rm min}$, and $p_{\rm cross}$, whose role will be explained later. See Figure 5.3 (b) for an example of the 2-3 tree whose leaves represent intervals of the NFP layout in Figure 5.3 (a).

For two nodes u and v of the tree, let $PATH(u, v)$ be the set of nodes in the path from u to v including u and v themselves. Let $g(k)$ be the overlap number for interval S_k of the sweep line. To be more precise, $g(k)$ is the overlap number of all points in S_k except the left (resp., right) boundary of S_k if it corresponds to the left (resp., right) edge of an

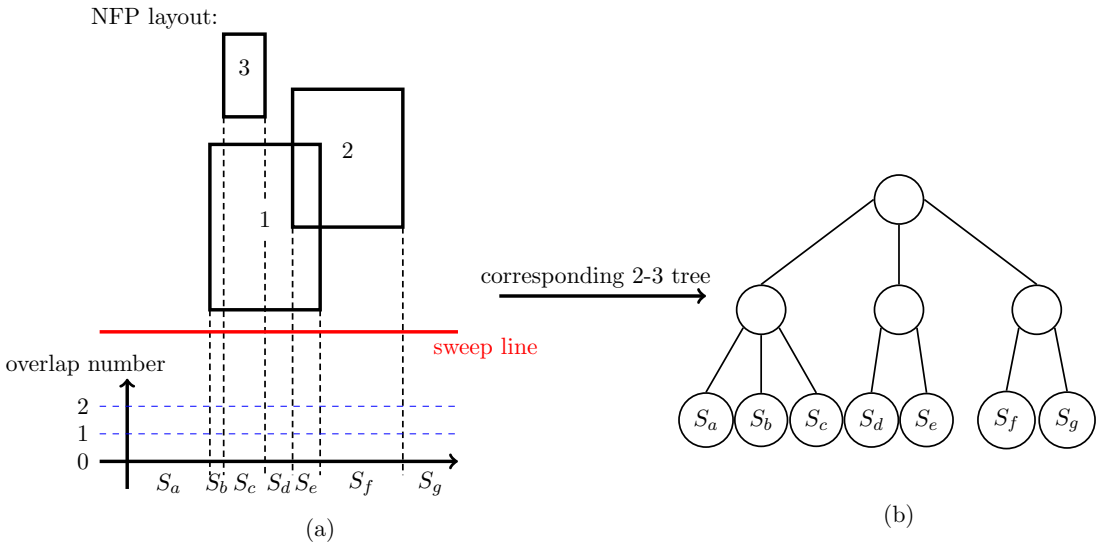


Figure 5.3: An example of intervals and corresponding 2-3 tree for an NFP layout

NFP rectangle. This means that the overlap number of the left (resp., right) edge of an NFP rectangle is stored in an interval S_k whose right (resp., left) boundary is that edge. Thus we need to treat the boundaries carefully considering that each NFP is an open set. When the value of y is fixed to the height of the current sweep line, the function $\pi(x, y)$ is a lower semi-continuous piecewise linear function consisting of horizontal line segments with heights $g(1), g(2), \ldots, g(2m_j^{\mathrm{T}}\tilde{M} + 7)$ aligned in this order from left to right. The algorithm maintains the values of p_{value} for all nodes of the tree so that

$$\sum_{u \in PATH(k, root)} p_{\mathrm{value}}(u) = g(k) \tag{5.2.1}$$

is satisfied for each leaf k, where *root* is the root node of the 2-3 tree. Then it is possible to compute the overlap number of an interval in $O(\log M)$ time using the values of p_{value} in the path from the corresponding leaf to the root node.

We define a *CrossPoint* as follows: If an NFP rectangle's right edge crosses another's top edge, we call the crossing point a CrossPoint. Figure 5.4 shows an example of the CrossPoints.

Observe that, a bottom-left stable feasible position will only appear at non-overlapping CrossPoints, and thus the BL position is also among them. To find such a CrossPoint efficiently, we prepare for each leaf k, a Boolean value $p_{\mathrm{cross}}(k)$ that takes value 1 if the left boundary of interval S_k is a right edge of an NFP rectangle whose interior intersects with the sweep line and 0 otherwise, and then we define the value of p_{min} for each node v

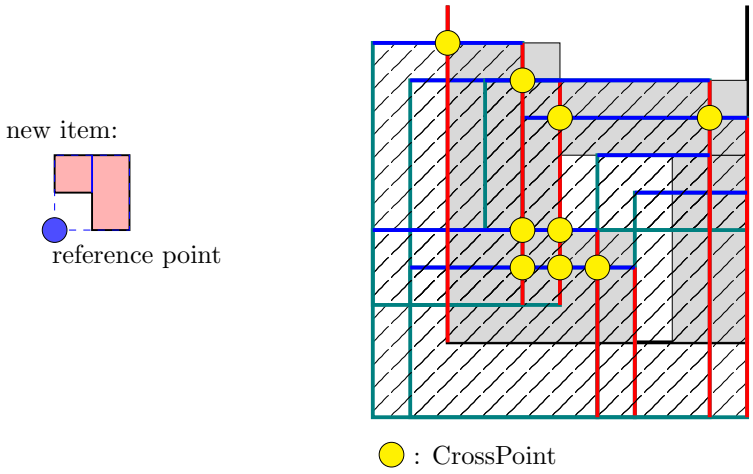


Figure 5.4: An example of CrossPoints of the new rectilinear block

of the 2-3 tree as follows:

$$p_{\min}(v) = \min_{k \in Q(v)} \sum_{u \in PATH(k,v)} p_{\text{value}}(u), \tag{5.2.2}$$

where $Q(v)$ is the set of all leaf nodes k such that $p_{\text{cross}}(k) = 1$ in the subtree rooted at the node v. For convenience, we assume $p_{\min}(v) = +\infty$ if $Q(v) = \emptyset$. Note that $p_{\min}(v) = +\infty$ indicates that there exists no leaf node whose left boundary corresponds to the right edge of an NFP rectangle with its interior intersecting with the sweep line among all leaf nodes in the subtree rooted at the node v. The value of $p_{\min}(v)$ is used to compute the minimum overlap number of intervals S_k corresponding to leaves k in $Q(v)$, which can be computed by

$$p_{\min}(v) - p_{\text{value}}(v) + \sum_{u \in PATH(v,root)} p_{\text{value}}(u). \tag{5.2.3}$$

Because the BL position will only appear at a CrossPoint, we can ignore the leaves not in $Q(v)$. Using the value of $p_{\min}(v)$ and the values of $p_{\text{value}}(u)$ for nodes u on the path from the parent node of v to the root node as shown in (5.2.3), it is possible to check whether there exists a leaf node in $Q(v)$ whose overlap number is equal to zero. Let a, b and c be the children of a node v. (Note that for a 2-3 tree, there are either two or three children for every node except the leaves. For simplicity, we assume that if the node v has just two children, the value of $p_{\min}(c) = +\infty$, which makes the node c having no effect on the final result.) Assume that the values of $p_{\min}(a)$, $p_{\min}(b)$ and $p_{\min}(c)$ are known. Then the

value of $p_{\min}(v)$ can be computed in constant time by

$$p_{\min}(v) = p_{\text{value}}(v) + \min\{p_{\min}(a),\ p_{\min}(b),\ p_{\min}(c)\}. \tag{5.2.4}$$

Consider the moment when the sweep line encounters an element in N_{tb}. Let B be the NFP rectangle whose top or bottom edge is encountered by the sweep line, and assume that the left (resp., right) edge of B is the lth (resp., $(r+1)$st) element in the sorted list of N_{lr}. In this situation, the overlap numbers for intervals $S_l, S_{l+1}, \ldots, S_r$ are changed. The overlap numbers of these intervals should be increase (resp., decrease) by one if the encountered edge is an element in N_{b} (resp., N_{t}).

We explain the algorithm to keep the values of p_{value}, $p_{\min}$ and p_{cross} appropriately, assuming that their values were correct before the sweep line encountered an edge. The algorithm first finds the leaf $r+1$ that corresponds to the $(r+1)$st interval. It then updates the value of $p_{\text{cross}}(r+1)$ to one (resp., zero) and the value of $p_{\min}(r+1)$ to $p_{\text{value}}(r+1)$ (resp., $+\infty$) if the bottom (resp,. top) edge of B is encountered. It further modifies the values of $p_{\min}$ of all nodes on the path from $r+1$ to the root by equation (5.2.4). The details of this procedure is formally described as algorithm UpdateCross(T, B) in Algorithm 9.

Algorithm 9 UpdateCross(T, B)

Input: A tree T and an NFP rectangle B whose right edge is the $(r+1)$st element in N_{lr}.
Output: Updated Tree T.

1: Find the leaf $r+1$ that corresponds to the $(r+1)$st interval.
2: Set $r' := r+1$, and $\lambda := 1$ (resp., -1) if the edge encountered by the sweep line is the bottom (resp., top) edge of B.
3: Set $p_{\text{cross}}(r') := 1$ and $p_{\min}(r') := p_{\text{value}}(r')$ if $\lambda = 1$; otherwise (i.e., $\lambda = -1$), set $p_{\text{cross}}(r') := 0$ and $p_{\min}(r') := +\infty$.
4: Modify the values of $p_{\min}$ of all nodes on the path from r' to the root by equation (5.2.4).

Then, the algorithm finds the leaves l and r that correspond to the lth and rth intervals. See Figure 5.5 as an example. Here we assume for simplicity that the edge encountered by the sweep line is the bottom edge of the B. The case when the top edge is encountered is similar; instead of increasing the values by one, the algorithm decreases the values by one. The algorithm increases the values of p_{value} and $p_{\min}$ of the leaf nodes l and r by one. It then traverses nodes on the paths from the leaf nodes l and r to their least common ancestor v. During this traversal, whenever a node in the path from l (resp., r) to v is reached from its left (resp., right) or middle child, the algorithm increases the values of p_{value} and $p_{\min}$ of all its right (resp., left) siblings by one. It also updates $p_{\min}$ for nodes

in the paths from l and r to v so that the condition (5.2.2) is satisfied (using equation (5.2.4)). Finally, the algorithm updates the values of $p_{\min}$ for all nodes in the path from v to the root node of the tree.

The details of this procedure is summarized as algorithm UpdateValue(T, B) in Algorithm 10. (Note that the depths of all leaves of a 2-3 tree are the same, and hence the least common ancestor of the leaves l and r can be found by going up the tree from l and r simultaneously. This feature is utilized in the description of Algorithm 10.)

Algorithm 10 UpdateValue(T, B)

Input: A tree T and an NFP rectangle B whose left and right edges are the lth and $(r+1)$st elements in N_{lr}.

Output: Updated Tree T.

1: Find the leaves l and r that corresponds to the lth and rth intervals.

2: Set $\lambda := 1$ (resp., -1) if the edge encountered by the sweep line is the bottom (resp., top) edge of B.

3: Invoke UpdateCross(T, B). Add the value λ to $p_{\mathrm{value}}(l)$, $p_{\min}(l)$, $p_{\mathrm{value}}(r)$ and $p_{\min}(r)$.

4: Let $l_{\mathrm{prev}} := l$ and $r_{\mathrm{prev}} := r$, and then let l be the parent of l, and r be the parent of r. If $l = r$, go to Step 4.

5: If the rightmost (resp., leftmost) child of l (resp., r) is different from l_{prev} (resp., r_{prev}), add the value λ to p_{value}, $p_{\min}$ of all right (resp., left) siblings of l_{prev} (resp., r_{prev}). Update the values of $p_{\min}(l)$ and $p_{\min}(r)$ by equation (5.2.4). Return to Step 2.

6: If the node $l(= r)$ has a child u between l_{prev} and r_{prev}, then add λ to $p_{\mathrm{value}}(u)$ and $p_{\min}(u)$. Then for each node v on the path from $l(= r)$ to the root, update the value of $p_{\min}(v)$ by equation (5.2.4). Stop.

Algorithm UpdateValue(T, B) runs in $O(\log M)$ time since the height of the 2-3 tree is $O(\log \tilde{M}) = O(\log M)$.

Lemma 5.2.1. *Assuming that M is the number of rectangles that represent all rectilinear blocks, procedure UpdateValue updates in $O(\log M)$ time the overlap number of intervals that are stored in a 2-3 tree when the sweep line encounters a bottom or top edge of an NFP rectangle.*

5.2.2 Search for BL position on a 2-3 tree

In this section, we propose an algorithm called *FindBL* to find the BL position of a shape T_j with respect to the packing layout of items in $\hat{R}$ using the 2-3 tree. As explained before, the BL position only appears at non-overlapping CrossPoints, where one NFP rectangle's

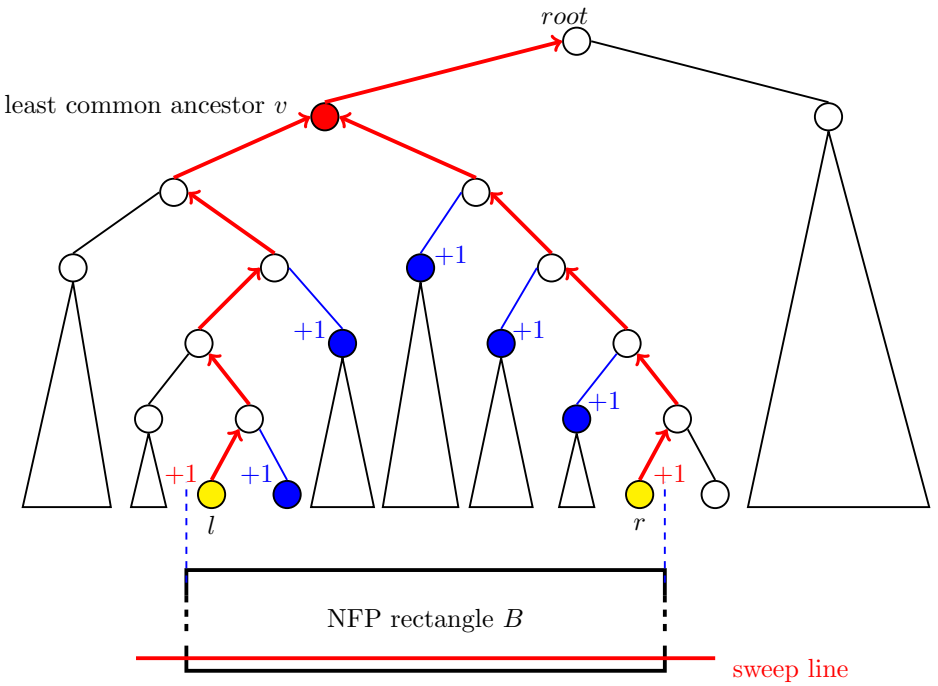


Figure 5.5: An example of procedure UpdateValue

right edge crosses another's top edge. Hence, when the sweep line encounters the top edge of an NFP rectangle, the UpdateValue algorithm modifies the overlap number, and the BL position may appear at this moment. To manage these events, the elements in N_{tb} are stored in $HEAP_j$ according to the non-decreasing order of their y-coordinates, where an element with smaller y-coordinate comes to the top of the heap, and ties are broken by putting more priority to elements in N_{t}. If the top edges of some NFP rectangles have the same y-coordinate, we put more priority to those elements that correspond to NFP rectangles whose left edge has a smaller x-coordinate.

A CrossPoint (x, y) whose overlap number $\pi(x, y)$ equals zero is a bottom-left stable feasible position of shape T_j relative to the current packing layout of $\hat{R}$. With the sweep line moving from bottom to top, the algorithm checks whether there exists such a point among all the CrossPoints on the sweep line whenever the sweep line encounters the top edge of an NFP rectangle. If there are more than one such point on the sweep line, the algorithm finds the leftmost one. The BL position is the first such point that the algorithm finds.

Our algorithm to calculate the BL position can deal with the situation where the

sweep line does not start from the bottom of the container. This feature is very important because the running time will be significantly reduced if we know a height $y_{\rm init}$ such that the BL position will not appear below the line $y = y_{\rm init}$ which is higher than the bottom of the container. Assume that we are given a 2-3 tree $TREE_j$ and a heap $HEAP_j$ that satisfy the following conditions.

C1. For the edge e in $N_{\rm t}$ with the highest priority (with respect to the ordering rule of elements in $N_{\rm tb}$) among those in $HEAP_j$, $HEAP_j$ contains all the elements in $N_{\rm tb}$ whose priority is lower than the edge e. (Note that e is not necessarily at the root of $HEAP_j$, because there may be edges in N_b having higher priority than that of e in the heap. If we keep deleting the element at the root of the heap, e is encountered first among the elements in $N_{\rm t}$.)

C2. The data structures $TREE_j$ and $HEAP_j$ are initialized so that when the FindBL algorithm is executed until the edge e in condition C1 is deleted from $HEAP_j$, the values on the nodes of the 2-3 tree $TREE_j$ keep the information of overlap numbers of the sweep line at the height of the edge e. To be more precise, at the time when e is deleted from $HEAP_j$, $TREE_j$ keeps the information of overlap numbers at the moment when the sweep line has passed all of the edges in $N_{\rm tb}$ whose priorities are higher than the edge at the root of $HEAP_j$.

Here we define the *BL order* '$\preceq_{\rm BL}$' between points in the plane by $(x, y) \preceq_{\rm BL} (x', y') \iff$ (1) $y < y'$ or (2) $y = y'$ and $x \leq x'$. Let $(x_{\rm init}, y_{\rm init})$ be the left endpoint of the edge e in condition C1. If the conditions C1 and C2 are satisfied, the FindBL algorithm outputs the smallest point with respect to $\preceq_{\rm BL}$ among the bottom-left stable feasible positions (x, y) that satisfies $(x_{\rm init}, y_{\rm init}) \preceq_{\rm BL} (x, y)$ and $(x, y) \neq (x_{\rm init}, y_{\rm init})$. Hence, if the following condition C3 is also satisfied, the FindBL algorithm outputs the BL position.

C3. The BL position does not exist below the line $y = y_{\rm init}$ nor on the half line $y = y_{\rm init}$ with $x \leq x_{\rm init}$, i.e., $\pi(x, y) > 0$ holds for every point $(x, y) \preceq_{\rm BL} (x_{\rm init}, y_{\rm init})$.

The FindBL algorithm first takes an element e at the root of $HEAP_j$ and removes it from the heap. Let $B \in \hat{\mathcal{R}}$ be the NFP rectangle having the element e as its top or bottom edge, and assume that its left (resp., right) edge is the left boundary of the lth (resp., the right boundary of rth) leaf of $TREE_j$. Then the algorithm updates the overlap numbers of intervals $S_l, S_{l+1}, \ldots, S_r$ by invoking the UpdateValue procedure. If e is a bottom edge, the algorithm proceeds to the next element at the top of the heap. Otherwise, it goes up the tree from the leaf l until it finds a right sibling u that satisfies

$$p_{\rm min}(u) - p_{\rm value}(u) + \sum_{v \in PATH(u, root)} p_{\rm value}(v) = 0, \tag{5.2.5}$$

i.e., the subtree rooted at u has a leaf k with $p_{\text{cross}}(k) = 1$ and the overlap number $g(k) = 0$. If the root is reached without finding such a sibling u, which implies that the current top edge e does not include a CrossPoint whose overlap number is zero, the algorithm proceeds to the next element at the root of $HEAP_j$. Otherwise, the algorithm chooses the leftmost sibling u that satisfies (5.2.5) among those to the right of the current node and then goes down the tree from u by choosing at each node the leftmost child that satisfies (5.2.5). If $k > r$ holds for the leaf k reached by the above procedure, which implies that the left boundary of interval S_k does not cross the edge e, the algorithm proceeds to the next element at the root of $HEAP_j$. Otherwise, the algorithm outputs the bottom-left stable feasible point (x, y), where x is the left boundary of S_k and y is the height of the current sweep line (i.e., the height of e). After the FindBL algorithm outputs the point (x, y), it inserts the top and bottom edges of the NFP rectangle B into $HEAP_j$. This is necessary for the next call to FindBL; the BL position may remain the same event or remain the same height after some NFP rectangles are added into the current NFP layout, and if this is the case, the same (x, y) or point with the same y-coordinate must be output in the next call.

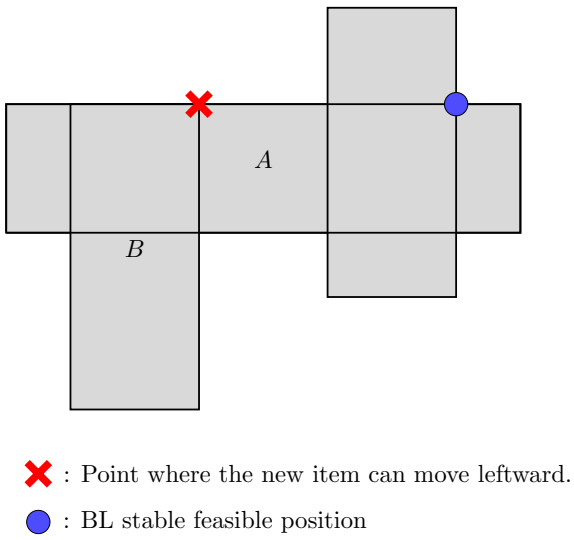


Figure 5.6: NFP rectangles whose top edges have the same y-coordinate

Care should be taken when the edge e overlaps with the element e' at the root of $HEAP_j$ after e is removed from the heap, and e' is also a top edge. If this happens, such top edges are merged into one longer edge with their leftmost (resp., rightmost) endpoint as its left (resp., right) endpoint, and the above procedure of traversing the tree is applied to this longer edge instead of the edge e. This is to avoid finding a wrong point in such

a case that is depicted in Figure 5.6, where the top edges of rectangles A and B have the same y-coordinate and A leaves the sweep line earlier. The wrong point of this case is the left point at the top right corner of B.

The details of our algorithm to calculate the BL stable feasible position next to $(x_{\text{init}}, y_{\text{init}})$ are summarized in Algorithm 11 as algorithm FindBL. In Algorithm 11, $l_{\min}$ (resp., $r_{\max}$) is the leaf corresponding to the left (resp., right) end of the merged edge, and *pathVal* keeps the sum of p_{value} for all nodes in the path from the current node to the root. The input of the FindBL algorithm consists of a 2-3 tree $TREE_j$ with $2m_j^{\mathrm{T}}\tilde{M} + 7$ leaves and a heap $HEAP_j$ that satisfy the two conditions C1 and C2. The algorithm outputs the BL stable feasible position next to $(x_{\text{init}}, y_{\text{init}})$.

Note that while Step 5 is repeated, the value of $\sum_{u \in PATH(\gamma, root)} p_{\text{value}}(u)$ is always zero, and it suffices to check if the value of $p_{\min}$ equals zero to check whether (5.2.5) is satisfied; hence it is not necessary to update *pathVal* once Step 5 is entered.

We now consider the time complexity of the FindBL algorithm. Let K be the number of elements deleted from $HEAP_j$ during the entire execution of a call to FindBL. Each call to Step 1 takes $O(1)$ time and Step 1 is called $O(K)$ times; hence the total execution time of Step 1 is $O(K)$.

By Lemma 5.2.1, procedure UpdateValue runs in $O(\log M)$ time. Then in Step 2, it takes $O(\log \tilde{M}) = O(\log M)$ time to delete an element from the heap, to find the leaves l and r, and to execute the UpdateValue procedure. Because Step 2 is called $O(K)$ times, the total execution time of this step is $O(K \log M)$.

In Step 3 and 4, the algorithm first climbs the 2-3 tree from the leaf $l_{\min}$, and whenever the current node α is not the rightmost child of its parent, it checks whether the subtree rooted at its right sibling α_{next} next to it has a leaf that contains a CrossPoint with overlap number zero. When the first node u having such a leaf is found, the algorithm sets $\gamma := u$.

In Step 5, the algorithm goes down the tree from γ, choosing the leftmost child including such a leaf that contains a CrossPoint with overlap number zero. Thus, once Step 3 is entered, the algorithm climbs up the tree from $l_{\min}$ in the loop of Step 3 and 4, and if it does not stop this traversal at the root, it goes down the tree to a leaf in Step 5. The time complexity of the loop of Step 3 and 4, once Step 3 is entered until the loop exits to Step 1 or 5, is $O(\log \tilde{M}) = O(\log M)$, because $O(\log \tilde{M})$ nodes are visited during the traversal from $l_{\min}$ to the γ when it exits to Step 5, and it is possible for each node u to check in constant time whether the subtree rooted at the node u has a leaf node that has a CrossPoint with overlap number zero. The time complexity of a loop of Step 5 from the time it is entered until it exits to Step 6 is also $O(\log M)$ for a similar reason. The loop of Steps 3 and 4 and that of Step 5 are entered at most K times, and hence the total time complexity of Step 3–5 is $O(K \log M)$.

Algorithm 11 FindBL($TREE_j$, $HEAP_j$, $(x_{\text{init}}, y_{\text{init}})$)

Input: A tree $TREE_j$, a heap $HEAP_j$ and a point $(x_{\text{init}}, y_{\text{init}})$.

Output: The BL stable feasible position next to $(x_{\text{init}}, y_{\text{init}})$.

1: Let $\lambda_{\text{prev}} := 1$.

2: If $HEAP_j$ is empty, output "no BL stable feasible position is found" and stop. Otherwise, let e be the element at the top of $HEAP_j$. Then let $B \in \hat{\mathcal{R}}$ be the NFP rectangle having the element e as its top or bottom edge, and assume that its left (resp., right) edge is the left boundary of the lth (resp., the right boundary of the rth) leaf of the 2-3 tree $TREE_j$. If e is the bottom (resp., top) edge of B, set $\lambda := 1$ (resp., -1). If (1) $\lambda_{\text{prev}} = -1$ and $\lambda = 1$ or (2) $\lambda_{\text{prev}} = -1$, $\lambda = -1$ and $r_{\max} + 1 \leq l$, then let $\alpha := l_{\min}$, $pathVal := \sum_{u \in PATH(\alpha, root)} p_{\text{value}}(u)$ and proceed to Step 3. If $\lambda_{\text{prev}} = 1$ and $\lambda = -1$, let $l_{\min} := l$ and $r_{\max} := r$. If $\lambda_{\text{prev}} = -1$ and $\lambda = -1$, let $r_{\max} := \max\{r_{\max}, r\}$. Invoke UpdateValue($T$, B). Remove the element e from $HEAP_j$, let $\lambda_{\text{prev}} := \lambda$, and then return to Step 2.

3: If α is the root, return to Step 1; otherwise, let α_{parent} be the parent node of α, and let $pathVal := pathVal - p_{\text{value}}(\alpha)$.

4: If α is the rightmost child of α_{parent}, set $\alpha := \alpha_{\text{parent}}$ and then return to Step 3; otherwise, set $\alpha := \alpha_{\text{next}}$ where α_{next} is the sibling next to α on the right. If the value of $pathVal + p_{\min}(\alpha) = 0$, then set $\gamma := \alpha$ and proceed to Step 5; otherwise return to Step 4.

5: If γ is a leaf, go to Step 6; otherwise, let γ_{child} be the leftmost child of γ among all children whose $p_{\min}$ equals 0. Let $\gamma := \gamma_{\text{child}}$ and then return to Step 5.

6: If $\gamma \geq r_{\max} + 1$, then return to Step 1. Otherwise, output (x, y), where x is the x-coordinate of the left boundary of the interval S_γ corresponding to the leaf γ and y is the y-coordinate of e. Add into $HEAP_j$ the top and bottom edges of a rectangle whose left (resp., right) edge corresponds to the left boundary of $l_{\min}$ (resp., the right boundary of $r_{\max}$), top edge is at the height y, and bottom edge is at an arbitrary height strictly smaller than y. Then stop.

The algorithm returns from Step 6 to 1 at most K times, and the latter part of Step 6 (i.e., after the "Otherwise") is executed only once, which takes $O(\log M)$ time. In summary, the total running time of algorithm FindBL is $O(K \log M)$.

Lemma 5.2.2. *Assuming that K is the number of elements removed from heap until the algorithm finds the BL position for a new rectilinear block, and M is the number of rectangles that represent all rectilinear blocks, the FindBL algorithm finds the BL position for the new item in $O(K \log M)$ time.*

In later sections, we show that the number of edges to be added to $HEAP_j$ is $O(m_j^{\mathrm{T}} M)$ during the entire execution of the bottom-left or best-fit algorithm. During the process of these heuristics, the FindBL algorithm is called many times, but the total number of edges deleted from $HEAP_j$ (i.e., the sum of the values of K for all calls to FindBL) is bounded by the number of added edges and hence is $O(m_j^{\mathrm{T}} M)$. This implies that the total execution time of FindBL for a shape T_j during the entire process of the bottom-left or best-fit algorithm is $O(m_j^{\mathrm{T}} M \log M)$.

Note that we can easily modify the FindBL algorithm to enumerate all the bottom-left stable feasible positions for a layout of rectilinear blocks. In Step 6 of Algorithm 11, instead of inserting the top and bottom edges of the rectangle whose left (resp., right) edge corresponds to the left boundary of $l_{\min}$ (resp., the right boundary of $r_{\max}$), we insert the top and bottom edges of a rectangle whose left (resp., right) edge corresponds to the left boundary of γ (resp., the right boundary of $r_{\max}$), where γ is the leaf that contains the bottom-left stable feasible position most recently found. With this modification, the algorithm outputs the bottom-left stable feasible position next to the one most recently found, instead of reporting the same bottom-left stable feasible position again. Accordingly, all we have to do to enumerate all bottom-left stable feasible positions is to call the modified FindBL algorithm iteratively until $HEAP_j$ becomes empty. Assume that κ is the number of bottom-left stable feasible positions. Then the modified FindBL algorithm is called $O(\kappa)$ times, and the time complexity of enumerating all such positions of a rectilinear shape that consists of m_j^{T} rectangles for a given layout of rectilinear blocks consisting of M rectangles is $O((m_j^{\mathrm{T}} M + \kappa) \log M)$, because $HEAP_j$ initially contains $O(m_j^{\mathrm{T}} M)$ elements and $O(\kappa)$ elements are added into it during the entire process of calling the modified FindBL iteratively.

Since the sweep line moves from bottom to top only once for each shape T_j, the entire process will not be affected even if we delete the NFPs whose top edges are strictly lower than the current sweep line. This idea can be implemented as follows. For each T_j, we maintain a queue that stores all the top edges removed from $HEAP_j$ in Step 2 of Algorithm 11. Then, whenever Algorithm 11 terminates, for every top edge e in the queue

that is strictly lower than the current sweep line, we delete from $TREE_j$ the two leaves l and $r+1$ corresponding to the intervals whose left boundaries are the left and right edges, respectively, of the NFP corresponding to e, modifying the right boundaries of $l-1$ and r, the leaves immediately to the left of l and $r+1$, to the right boundaries of l and $r+1$, respectively (of course information in the inner nodes above the modified leaves should be modified appropriately). Because the number of edges to be deleted from $HEAP_j$ cannot be more than those inserted to it, and it takes $O(\log M)$ time to delete or modify a leaf, the above process for a shape T_j takes $O(m_j^{\mathrm{T}} M \log M)$ time during the entire packing process. This implies that this deleting process has no negative effect on the time complexity of algorithm FindBL, while it may significantly reduce the memory space in practice.

Theorem 5.2.3. *Assuming that M is the number of rectangles that represent all rectilinear blocks, and m_j^{T} is the number of rectangles that represent a rectilinear shape T_j, the modified FindBL algorithm enumerates all the bottom-left stable feasible positions for T_j in $O(m_j^{\mathrm{T}} M \log M)$ time.*

5.3 Initialization and modification of trees and heaps

In this section, we explain how we modify the data structure dynamically and then give the function to initialize the trees and heaps.

For each shape, our algorithms need to dynamically keep the NFP layout with respect to the current packing layout during the process. We give a function to modify the 2-3 tree and the heap when an NFP rectangle B is added into the NFP layout.

The function first inserts the top and bottom edges of B into $HEAP_j$ (according to the priority among the elements in N_{tb} explained at the beginning of Section 5.2.2). It then finds the leaf k such that the corresponding interval S_k contains the left edge l of B (according to the ordering of the elements in N_{lr} explained in Section 5.2.1), and it divides the leaf k into two leaves k_1 and k_2 corresponding to intervals S_{k_1} and S_{k_2}, respectively, where the left (resp., right) boundary of S_{k_1} (resp., S_{k_2}) is that of S_k, and the right (resp., left) boundary of S_{k_1} (resp., S_{k_2}) is the edge l. Then, the values of p_{value}, p_{cross} and p_{min} of the leaf k are copied to the new leaves k_1 and k_2. The rebalance operator of the 2-3 tree is then invoked so that the resulting tree satisfies the conditions that must be satisfied by a 2-3 tree (see the Remark at the end of this subsection). The right edge of B is processed similarly. The procedure is summarized as function ModifyTH in Algorithm 12.

The computation time of function ModifyTH is $O(\log m_j^{\mathrm{T}} \tilde{M}) = O(\log M)$, where $m_j^{\mathrm{T}} \tilde{M}$ is the number of NFP rectangles in the NFP layout when procedure ModifyTH is invoked.

Algorithm 12 ModifyTH(B, $TREE_j$, $HEAP_j$)

Input: An NFP rectangle B, a tree $TREE_j$ and a heap $HEAP_j$.

Output: Updated $TREE_j$ and $HEAP_j$.

1: Let l, r, b and t be the left, right, bottom and top edge of the NFP rectangle B.
2: Insert b and t into $HEAP_j$. Let $e := l$.
3: Find the leaf k that corresponds to the interval S_k where the edge e should be inserted according to the ordering rule of elements in N_{lr} explained in Section 5.2.1.
4: Divide the leaf k into two leaf nodes k_1 and k_2 corresponding to two intervals S_{k_1} and S_{k_2}. Set the left (resp., right) boundary of S_{k_1} (resp., S_{k_2}) to that of S_k, and set the right (resp., left) boundary of S_{k_1} (resp., S_{k_2}) to the edge e. For each $i = 1$ and 2, let $p_{\mathrm{value}}(k_i) := p_{\mathrm{value}}(k)$, $p_{\mathrm{cross}}(k_i) := p_{\mathrm{cross}}(k)$ and $p_{\mathrm{min}}(k_i) := p_{\mathrm{min}}(k)$.
5: Rebalance the 2-3 tree.
6: If $e = r$, then stop. Otherwise, let $e := r$ and return to Step 3.

Lemma 5.3.1. *The procedure ModifyTH updates in $O(\log M)$ time the corresponding 2-3 tree and heap when inserting an NFP rectangle into the container, where $m_j^{\mathrm{T}}\tilde{M}$ is the number of NFP rectangles in the NFP layout.*

At the beginning of the bottom-left or best-fit algorithm, no items are placed in the container (except for the four container rectangles). Corresponding to this empty layout, for each shape T_j, we prepare a 2-3 tree $TREE_j$ with seven leaves corresponding to the intervals defined by the left and right edges of the NFPs of the bounding box of T_j relative to the container rectangles and a heap $HEAP_j$ with eight elements consisting of the bottom and the top edges of the NFPs (note that each of these NFPs is a rectangle). See Figure 5.7 for an example of intervals corresponding to the empty layout and an item in (a). In Figure 5.7 (b), NFP_i ($i = 1, \ldots, 4$) signifies the NFP of the bounding box of T_j relative to container rectangle C_i in $\mathcal{C}$.

The elements in the heap are arranged according to the priority rule among the top and bottom edges explained at the beginning of Section 5.2.2, and the leaves of the tree are ordered from left to right according to the ordering rule of the left and right edges explained in Section 5.2.1. Note that at this moment, the sweep line is at a sufficiently low position whose y-coordinate is lower than the bottom edges of all container rectangles. As a result, the values of p_{value}, p_{min} and p_{cross} of all nodes are 0. The initialization phase is summarized as procedure InitializeTH in Algorithm 13.

As explained before, the number of rectangles that represents a rectilinear shape T_j is denoted by m_j^{T}, and the sum of m_j^{T} over t distinct shapes is denoted by m. Because it takes $O(m_j^{\mathrm{T}})$ time to compute the NFP of T_j relative to a container rectangle (even though

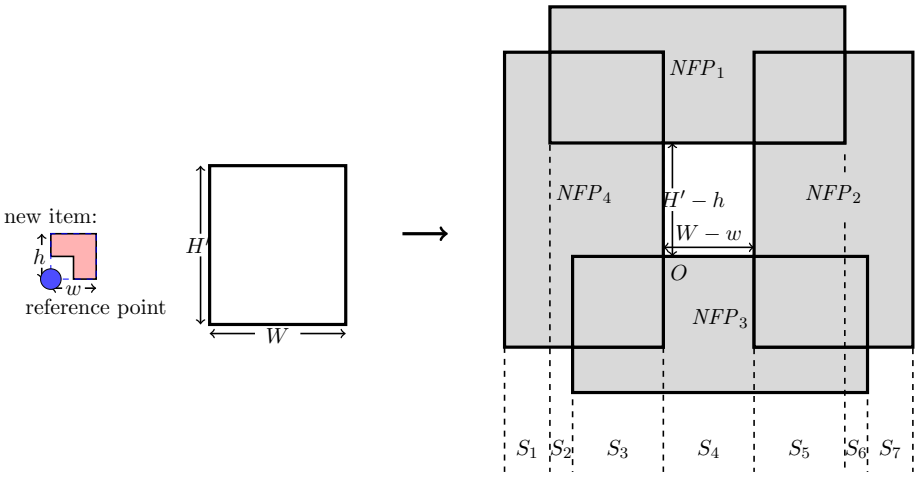


Figure 5.7: Intervals corresponding to the empty layout

Algorithm 13 InitializeTH

Input: Four container rectangles and shapes $\{T_1, T_2, \ldots, T_t\}$.

Output: $TREE_j$ and $HEAP_j$ for each shape T_j.

For every shape T_j $(j = 1, 2, \ldots, t)$, do the following Steps 1 and 2.

1: Compute the NFPs of T_j relative to the four container rectangles in $\mathcal{C}$.

2: Create a heap $HEAP_j$ that consists of the top and bottom edges of the four NFPs computed in Step 1. Then create a 2-3 tree $TREE_j$ that has seven leaves corresponding to the seven intervals defined by the left and right boundaries of the four NFPs. Set the values of p_{value} and $p_{\min}$ to zero for all nodes in the tree, and set the values of p_{cross} to zero for all leaves of the tree.

such an NFP is a rectangle), Step 1 takes $O(m_j^{\mathrm{T}})$ time for each j. The sizes of $HEAP_j$ and $TREE_j$ are $O(1)$, and hence Step 2 takes $O(1)$ time for each j. This initialization phase therefore runs in $\sum_{j=1}^{t} O(m_j^{\mathrm{T}}) = O(m)$ time.

At the beginning of the bottom-left or best-fit algorithm, the InitializeTH procedure is called, and whenever an NFP rectangle is placed, the ModifyTH procedure is executed. The FindBL algorithm is invoked whenever needed. The data structures $TREE_j$ and $HEAP_j$ are modified only by these three procedures. It is not hard to see that the conditions C1–C3 in Section 5.2.2 are satisfied after procedure InitializeTH has finished and they are satisfied whenever the ModifyTH or FindBL algorithm terminates if they are satisfied before the algorithm is invoked. These three conditions are therefore satisfied whenever FindBL begins its computation. Consequently, the FindBL algorithm correctly computes the BL position whenever it is called.

Theorem 5.3.2. *The FindBL algorithm correctly computes the BL position of a new rectilinear block.*

Remark. A 2-3 tree must satisfy the following two conditions:

- All leaves are at the same depth.
- Every inner node has two or three children.

When a leaf k is divided into two leaf nodes k_1 and k_2, the number of children of the parent α of k is increased by one, and α may have four children. If this is the case, α is divided into two nodes α_1 and α_2, where α_1 has the first two children of α, α_2 has the latter two, and both have the same parent as α. The value of $p_{\mathrm{value}}(\alpha)$ is copied to α_1 and α_2, and the values of $p_{\mathrm{min}}(\alpha_1)$ and $p_{\mathrm{min}}(\alpha_2)$ are computed by (5.2.4). If the parent of α has four children, the same operation is applied to it, and this process is repeated by going up the tree until there is no node having four children. Note that when the root is divided into two nodes, a new root having them as its children is created. We explained similar operation on a 2-3 tree when inserting a new leaf in Section 3.2.2. See Figure 3.4 as an example. Thus, the rebalance operation is processed by moving up the tree and hence can be done in time proportional to the height of the tree, i.e., $O(\log(m_j^{\mathrm{T}}\tilde{M})) = O(\log M)$.

5.4 Construction heuristics with efficient implementations

In this section, we explain efficient implementations of construction heuristics for the rectilinear block packing problem in which the FindBL algorithm and relevant data structures and procedures are utilized. We first explain the efficient implementations of the

bottom-left and best-fit algorithms in Section 5.4.1 and 5.4.2. We then explain their time complexity in Section 5.5.

5.4.1 Bottom-left algorithm based on FindBL algorithm

The bottom-left algorithm can be generally explained as follows: Given a set of n rectilinear blocks $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and an order of items (e.g., decreasing order of area), the algorithm packs all the items one by one according to the given order, where each item is placed at its BL position relative to the current layout (i.e., the layout at the time just before it is placed).

Assume for simplicity that $\{R_1, R_2, \ldots, R_n\}$ are packed according to the increasing order of their indices. The bottom-left algorithm, in which FindBL algorithm is utilized to find BL positions, is formally described as Algorithm 14.

Algorithm 14 Bottom-Left_FindBL

Input: A sequence of rectilinear blocks $R_1, R_2, \ldots, R_n$ and a container.

Output: A packing layout.

1: Call procedure InitializeTH. Let $i := 0$.

2: Set $i := i + 1$. If $i > n$, output the packing layout and stop. Let $TREE_j$ and $HEAP_j$ be the 2-3 tree and heap corresponding to the shape T_j of item R_i.

3: Call algorithm FindBL($TREE_j$, $HEAP_j$, $(0, 0)$) to find the BL position (x, y) of R_i.

4: Pack R_i at (x, y).

5: For each shape T_j $(j = 1, 2, \ldots, t)$, call procedure ModifyTH(B, $TREE_j$, $HEAP_j$) for every NFP rectangle B that constitutes $NFP(R_i, T_j)$ to add the NFP of T_j relative to R_i into the corresponding NFP layout. Then return to Step 2.

5.4.2 Best-fit algorithm based on FindBL algorithm

Assume that the rectilinear blocks are divided into groups according to the shape and we are given a sequence of items of each shape that represents the decreasing order of their priority. In each iteration, we compute the BL positions for all of the remaining shapes using FindBL, and then we choose the rectilinear shape whose BL position takes the smallest x-coordinate among those with the lowest y-coordinate, breaking ties by choosing the shape having a remaining item with the highest priority. Then we take the item at the top of the ordered list of the chosen shape and pack it into the container. We then insert the NFPs of this item into the NFP layout for each shape and update the trees and heaps. The best-fit algorithm is formally described as Algorithm 15.

Algorithm 15 Best-Fit_FindBL

Input: A set of rectilinear blocks $\mathcal{R}$ and a container.

Output: A packing layout.

1: Set $\mathcal{R}' := \mathcal{R}$.

2: If $\mathcal{R}' = \emptyset$, output the packing layout and stop.

3: Set $\mathcal{T}' := \mathcal{T}$, $x^* := +\infty$, $y^* := +\infty$, $x_{\text{init}} := 0$ and $y_{\text{init}} := 0$.

4: Choose a shape T_j in $\mathcal{T}'$, and then let $\mathcal{T}' := \mathcal{T}' \setminus \{T_j\}$. If there is no item in $\mathcal{R}'$ whose shape is T_j, return to Step 4.

5: Find the BL position (x, y) of T_j using FindBL($TREE_j$, $HEAP_j$, $(x_{\text{init}}, y_{\text{init}})$). Let $R_i \in \mathcal{R}'$ be the item that takes the highest priority among those items whose shape is T_j. If one of the following three conditions holds, then let $x^* := x$, $y^* := y$ and $i^* := i$.

(i) $y < y^*$.

(ii) $y = y^*$ and $x < x^*$.

(iii) $y = y^*$, $x = x^*$ and R_i has higher priority than R_{i^*}.

6: If $\mathcal{T}' \neq \emptyset$, return to to Step 4.

7: Pack the item R_{i^*} at (x^*, y^*).

8: For each shape T_j $(j = 1, 2, \ldots, t)$, call procedure ModifyTH(B, $TREE_j$, $HEAP_j$) for every NFP rectangle B that constitutes $NFP(R_{i^*}, T_j)$ to add the NFP of T_j relative to R_{i^*} into the corresponding NFP layout.

9: Set $\mathcal{R}' := \mathcal{R}' \setminus \{R_{i^*}\}$, $x_{\text{init}} := x^*$, $y_{\text{init}} := y^*$, and return to Step 2.

5.5 Time complexities

In this section, we explain the time complexities of the new implementation of the heuristic algorithms for the rectilinear block packing problem.

For every iteration, our algorithms pack a rectilinear block into the container. When a block R_i is placed, $m_i m_j^{\mathrm{T}}$ NFP rectangles are placed into the NFP layout of shape T_j, where m_i and m_j^{T} are the numbers of rectangles that represent R_i and T_j, respectively.

By Lemma 5.3.1, procedure ModifyTH takes $O(\log M)$ time for each NFP rectangle. Time for modifying the balanced search trees and heaps by calling procedure ModifyTH for all shapes is $\sum_{j=1}^{t} O(m_i m_j^{\mathrm{T}} \log M) = O(m_i m \log M)$, where $m = \sum_{j=1}^{t} m_j^{\mathrm{T}}$. Because we need to insert such NFP rectangles into the NFP layout of every shape T_j for all items R_i in $\mathcal{R}$, the time spent for procedure ModifyTH during the entire execution of the bottom-left or best-fit algorithm is $\sum_{i=1}^{n} O(m_i m \log M) = O(mM \log M)$.

According to Lemma 5.2.2, the execution time of a call to the FindBL algorithm is $O(K \log M)$ for K the number of elements deleted from $HEAP_j$ during the call to FindBL. For convenience, let K_{jl} be the number of elements deleted from $HEAP_j$ by the call to FindBL in the lth iteration of the bottom-left or best-fit algorithm. The number of deleted elements never exceeds the number of elements added to it. The number of elements added to $HEAP_j$ in the initialization phase is $O(1)$, and that of the iteration when an item R_i is added is $2m_i m_j^{\mathrm{T}} + 2$ (the top and bottom edges of $m_i m_j^{\mathrm{T}}$ NFP rectangles, and the top and bottom edges added into the heap in Step 6 of FindBL). Hence the total number of elements added into $HEAP_j$ is $\sum_{i=1}^{n}(2m_i m_j^{\mathrm{T}} + 2) + O(1) = O(m_j^{\mathrm{T}} M)$, which implies that $\sum_{l=1}^{n} K_{jl} = O(m_j^{\mathrm{T}} M)$. As a result, the total running time for computing BL positions of shape T_j by FindBL is $\sum_{l=1}^{n} O(K_{jl} \log M) = O(m_j^{\mathrm{T}} M \log M)$. Hence the total time of FindBL for all shapes is $\sum_{j=1}^{t} O(m_j^{\mathrm{T}} M \log M) = O(mM \log M)$ during the entire execution of the bottom-left or best-fit algorithm.

The time complexity of other parts of the algorithms are dominated by the execution time of these two procedures. Therefore, both the bottom-left and best-fit algorithms run in $O(mM \log M)$ time.

Theorem 5.5.1. *Assuming that M is the number of rectangles that represent all rectilinear blocks and m is the number of rectangles that represent all distinct shapes of rectilinear blocks, both the bottom-left and the best-fit algorithms run in $O(mM \log M)$ time using the FindBL algorithm.*

Note that it is not necessary to add in an NFP layout an NFP rectangle whose top edge is strictly lower than the current sweep line, because the sweep line never moves downward and hence such an NFP rectangle will not affect the overlap number during the

remaining execution of the bottom-left or best-fit algorithm. According to the definitions of reference points and NFP, the highest edge of $NFP(R_i, T_j)$ of two rectilinear blocks R_i and T_j is not higher than the top edge of the bounding box of R_i. Based on this, in Step 4 and Step 7 of Algorithm 14 and 15, respectively, after packing the item R_i or R_{i^*} into container, we first check the top edge of its bounding box. For each shape, if it is lower than the corresponding sweep line, there is no need to insert its NFP rectangles into the NFP layout. Otherwise, for every NFP rectangle of the item R_i or R_{i^*} that has just been packed, we check its top edge and insert it into the NFP layout only when it is not lower than the sweep line. Even with this modification, the worst-case time complexities of the two algorithms are the same, but this usually reduces the running time and memory space in practice.

5.6 Computational results

The bottom-left and best-fit algorithms proposed in Chapter 4 and 5 were implemented in the C programming language and run on a Mac PC with a 2.3 GHz Intel Core i5 processor and 4 GB memory.

The performance of these algorithms have been tested on a series of instances, which are generated from nine benchmark instances. The information of these benchmark instances is addressed in Table 5.1. For more details of these instances, readers could refer to [21]. We generate a set of instances named *C-class* by copying every shape in these nine instances. Table 5.2 shows the information of these classes of instances. For example, the set of instances generated with this rule from the instance "ami49L21" is labeled "C-ami49L21." The width W of the container is set to $W = \left\lceil \sqrt{\sum_{i=1}^{n} A(R_i)} \right\rceil$ according to the total sum of areas of items. The column of "#inst" reports the number of instances in each class. All of these instances are available at http://www.co.cm.is.nagoya-u.ac.jp/~yagiura/rectilinear/.

We analyze the computational results from both sides of the running time and the occupation rate. As to the order of the items for the bottom-left algorithm and the priority among the items for the best-fit algorithm, we tested the decreasing order of width, height, area and the area of the bounding box. The occupation rate of the decreasing order of area is slightly better than the results obtained by other orders. Hence, we report those results of the decreasing order of area.

The computational results obtained by the bottom-left and the best-fit algorithm are shown in Table 5.3 to 5.11. Each table shows the results for an instance set in C-class. In the tables, the columns of "Occup." of "Bottom-Left" and "Best-Fit" show the occupation rate in % obtained by the bottom-left and the best-fit algorithms. For each instance, the best results among the two algorithms are marked by '$*$'. The column "FBLR" is the

running time of the algorithms explained in Chapter 4, which utilize the Find2D-BL-R to find the BL positions. The column "FindBL" is the running time of the algorithms with efficient implementations in Chapter 5. All the running times are shown in seconds.

The computational results in column "FBLR," which are obtained by the bottom-left and best-fit algorithms explained in Chapter 4, show that the running time of the best-fit algorithm is much smaller than that of the bottom-left algorithm. The computational results in columns "FBLR" and "FindBL" show that our efficient implementations significantly reduce the computation time of the bottom-left and best-fit algorithms. By incorporating FindBL, the bottom-left algorithm becomes more than 500 times faster for the instance B10_2048, and the best-fit algorithm becomes more than 30 times faster for B10_2048 and T64_1024.

Observe in columns "FBLR" that the running time of the best-fit algorithm that utilizes the Find2D-BL-R algorithm to compute BL positions is much smaller for large-scale instances than that of the bottom-left algorithm. As explained in Section 4.4, in every iteration, the best-fit algorithm can discard the space below the sweep line where the latest item is placed. This significantly reduces the computation time for large-scale instances.

As to the occupation rate, the data in column "Occup." show that our algorithms tend to perform better when the instance becomes larger. Which of the bottom-left and best-fit algorithms is better depends on instances, and if we take the best result of these two, the occupation rate often reaches higher than 95%.

Table 5.1: Information of benchmark instances

Name	t	n	m	M
ami49L21	28	28	49	49
ami49LT21	27	27	49	49
TMCNCGSRC	51	51	76	76
B10	9	9	14	14
B30	29	29	59	59
T19	19	19	42	42
T40	32	32	42	42
T64	15	15	33	33
T144	20	20	31	31

Table 5.2: Information of C-class instances that are generated by copying benchmark instances

Name	#inst	t	n	m	M	W
C-ami49L21	10	28	28–14336	49	49–25088	5936–134337
C-ami49LT21	10	27	27–13924	49	49–25088	5953–134714
C-TMCNCGSRC	9	51	51–13056	76	76–19456	2376–38018
C-B10	12	9	9–18432	14	14–28672	9–438
C-B30	10	29	29–14848	59	59–30208	29–676
C-T19	11	19	19–19456	42	42–43008	18–601
C-T40	10	32	32–16384	42	42–21504	301–6818
C-T64	11	15	15–15360	33	33–33792	7–249
C-T144	10	20	20–10204	31	31–15872	11–249

Table 5.3: Computational results for ami49L21 (28 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
ami49L21_001	5936	28	49	49	*85.00	0.0000	0.0039	84.66	0.0068	0.0039
ami49L21_002	8396	56	49	98	83.41	0.0084	0.0078	*86.42	0.0185	0.0090
ami49L21_004	11873	112	49	196	*86.91	0.0294	0.0173	86.55	0.0536	0.0215
ami49L21_008	16792	224	49	392	*87.68	0.1412	0.0374	86.98	0.1538	0.0504
ami49L21_016	23747	448	49	784	*88.40	0.5864	0.1054	87.17	0.4213	0.1243
ami49L21_032	33584	896	49	1568	85.31	2.5567	0.2384	*87.78	1.2249	0.2770
ami49L21_064	47495	1792	49	3136	*89.49	10.9921	0.5121	87.82	3.4498	0.5632
ami49L21_128	67168	3584	49	6272	*89.93	47.3732	1.1610	88.28	10.1575	1.2505
ami49L21_256	94991	7168	49	12544	*90.19	218.3621	2.3410	88.84	29.1903	3.1304
ami49L21_512	134337	14336	49	25088	*90.59	1026.5724	5.0132	88.66	84.3987	7.6326

Table 5.4: Computational results for ami49LT21 (27 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
ami49LT21_001	5953	27	49	49	82.58	0.0002	0.0055	*86.80	0.0063	0.0061
ami49LT21_002	8419	54	49	98	84.71	0.0120	0.0085	*87.42	0.0185	0.1190
ami49LT21_004	11907	108	49	196	*87.14	0.0312	0.0188	86.09	0.0541	0.2697
ami49LT21_008	16839	216	49	392	*88.57	0.1245	0.0472	85.31	0.1492	0.5101
ami49LT21_016	23814	432	49	784	*87.96	0.5812	0.0948	86.61	0.4444	0.1184
ami49LT21_032	33678	864	49	1568	*88.44	2.4679	0.2165	87.54	1.2210	0.2553
ami49LT21_064	47628	1728	49	3136	*89.27	10.5217	0.5110	87.41	3.7520	0.6144
ami49LT21_128	67357	3456	49	6272	*89.96	43.1320	1.2052	87.70	9.8772	1.3976
ami49LT21_256	95257	6912	49	12544	*89.65	216.9078	2.6589	87.73	27.5558	3.1103
ami49LT21_512	134714	13824	49	25088	*90.00	1012.3210	5.9850	88.38	80.0347	6.9339

Table 5.5: Computational results for TMCNCGSRC (51 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
TMCNCGSRC_001	2376	51	76	76	*83.85	0.0015	0.0109	73.68	0.0264	0.0127
TMCNCGSRC_002	3360	102	76	152	*84.87	0.0194	0.0224	78.47	0.0813	0.0268
TMCNCGSRC_004	4752	204	76	304	*85.60	0.0623	0.0485	79.70	0.2348	0.0627
TMCNCGSRC_008	6720	408	76	608	*88.37	0.3134	0.1176	81.71	0.6242	0.1492
TMCNCGSRC_016	9504	816	76	1216	*86.94	1.3324	0.2757	83.03	1.9218	0.3489
TMCNCGSRC_032	13441	1632	76	2432	*88.05	5.7123	0.5914	83.97	5.3972	0.7934
TMCNCGSRC_064	19009	3264	76	4864	*88.44	24.5076	1.3641	84.72	15.6673	1.9113
TMCNCGSRC_128	26882	6528	76	9728	*88.06	106.6426	3.1013	84.75	45.0612	4.1709
TMCNCGSRC_256	38018	13056	76	19456	*88.12	484.3105	6.7650	85.38	131.3252	9.4778

Table 5.6: Computational results for B10 (9 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
B10_0001	9	9	14	14	*87.04	0.0004	0.0006	*87.04	0.0005	0.0004
B10_0002	13	18	14	28	*80.34	0.0018	0.0013	*80.34	0.0017	0.0010
B10_0004	19	36	14	56	*82.46	0.0097	0.0015	79.16	0.0049	0.0026
B10_0008	27	72	14	112	84.40	0.0194	0.0048	*89.84	0.0129	0.0054
B10_0016	38	144	14	224	87.95	0.0456	0.0073	*89.95	0.0372	0.0080
B10_0032	54	288	14	448	87.04	0.1645	0.0188	*92.84	0.0100	0.0153
B10_0064	77	576	14	896	88.78	0.7894	0.0311	*93.01	0.2929	0.0338
B10_0128	109	1152	14	1792	89.74	2.9940	0.0710	*94.35	0.7383	0.0682
B10_0256	155	2304	14	3584	90.79	12.6032	0.1326	*93.53	2.2887	0.1580
B10_0512	219	4608	14	7168	90.44	55.2214	0.3142	*93.52	6.9335	0.3511
B10_1024	310	9216	14	14336	91.32	244.6453	0.7025	*93.81	19.1195	0.8663
B10_2048	438	18432	14	28672	91.57	834.0140	1.6130	*96.39	57.6938	1.8165

Table 5.7: Computational results for B30 (29 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
B30_001	29	29	59	59	79.05	0.0001	0.0077	*83.32	0.0108	0.0069
B30_002	42	58	59	118	81.87	0.0091	0.0142	*85.41	0.0266	0.0157
B30_004	59	116	59	236	84.18	0.0406	0.0343	*86.59	0.0949	0.0388
B30_008	84	232	59	472	84.30	0.2105	0.0708	*88.69	0.2971	0.0867
B30_016	119	464	59	944	84.65	0.9015	0.1507	*89.70	0.8364	0.2009
B30_032	169	928	59	1888	85.06	3.8053	0.3360	*89.57	2.3240	0.4642
B30_064	239	1856	59	3776	85.19	16.4675	0.7607	*90.34	6.9546	1.0543
B30_128	338	3712	59	7552	86.15	71.1954	1.7002	*90.28	20.3806	2.3403
B30_256	478	7424	59	15104	86.27	323.4450	3.8178	*90.17	56.1533	5.2021
B30_512	676	14848	59	30208	87.03	1470.6230	8.6587	*90.64	161.9050	11.5104

Table 5.8: Computational results for T19 (19 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
T19_0001	18	19	42	42	*67.62	0.0003	0.0042	65.37	0.0049	0.0049
T19_0002	26	38	42	84	*73.39	0.0018	0.0081	69.63	0.0159	0.0078
T19_0004	37	76	42	168	72.00	0.0168	0.0162	*74.83	0.0466	0.0161
T19_0008	53	152	42	336	72.99	0.0923	0.0284	*75.05	0.1389	0.0330
T19_0016	75	304	42	672	74.56	0.4012	0.0623	*76.84	0.3776	0.0756
T19_0032	106	608	42	1344	75.05	1.6874	0.1464	*78.94	1.1887	0.1880
T19_0064	150	1216	42	2688	75.69	7.2680	0.3188	*79.27	3.2077	0.3873
T19_0128	212	2432	42	5376	75.05	30.8377	0.7202	*78.94	9.7410	0.9248
T19_0256	300	4864	42	10752	75.69	138.2885	1.5685	*79.48	27.4225	1.9829
T19_0512	425	9728	42	21504	74.87	610.6031	3.4406	*80.39	79.1770	4.0719
T19_1024	601	19456	42	43008	74.81	2817.6456	8.4418	*79.87	239.7320	9.0745

Table 5.9: Computational results for T40 (32 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
T40_001	301	32	42	42	*88.72	0.0004	0.0041	81.53	0.0080	0.0052
T40_002	426	64	42	84	*90.70	0.0018	0.0069	87.00	0.0237	0.0095
T40_004	602	128	42	168	*92.82	0.0214	0.1237	91.41	0.0667	0.0171
T40_008	852	256	42	336	92.67	0.0841	0.0324	*93.69	0.1817	0.0338
T40_016	1205	512	42	672	*94.19	0.3924	0.0687	92.74	0.5261	0.0802
T40_032	1704	1024	42	1344	*95.26	1.6252	0.1529	93.18	1.5402	0.2013
T40_064	2410	2048	42	2688	*96.07	6.8731	0.3383	92.74	4.2180	0.4619
T40_128	3409	4096	42	5376	*96.58	29.0134	0.7906	93.15	11.9671	1.0504
T40_256	4821	8192	42	10752	*97.01	128.0145	1.7350	93.44	33.2296	2.3315
T40_512	6818	16384	42	21504	*96.86	576.5932	3.9335	93.53	99.5149	5.1017

Table 5.10: Computational results for T64 (15 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
T64_0001	7	15	33	33	79.22	0.0001	0.0017	*87.14	0.0028	0.0028
T64_0002	11	30	33	66	*85.31	0.0006	0.0055	*85.31	0.0088	0.0053
T64_0004	15	60	33	132	85.61	0.0143	0.0092	*90.37	0.0280	0.0109
T64_0008	22	120	33	264	85.31	0.0678	0.0180	*92.42	0.0866	0.0184
T64_0016	31	240	33	528	87.46	0.2453	0.0380	*92.60	0.2469	0.0400
T64_0032	44	480	33	1056	90.54	1.0780	0.0839	*94.39	0.7050	0.1128
T64_0064	62	960	33	2112	91.26	4.5690	0.1867	*93.98	2.0836	0.2161
T64_0128	88	1920	33	4224	92.42	19.3126	0.4416	*95.41	6.0466	0.4700
T64_0256	124	3840	33	8448	93.29	86.1563	0.9837	*96.87	17.6114	1.0678
T64_0512	176	7680	33	16896	93.40	382.6577	2.1363	*96.44	51.6020	2.3696
T64_1024	249	15360	33	33792	93.26	1692.5672	4.7761	*97.23	153.8780	5.0661

Table 5.11: Computational results for T144 (20 distinct shapes)

					Bottom-Left			Best-Fit		
Instance	W	n	m	M	Occup.	FBLR	FindBL	Occup.	FBLR	FindBL
T144_001	11	20	31	31	85.31	0.0000	0.0028	*92.42	0.0029	0.0025
T144_002	15	40	31	62	85.61	0.0010	0.0048	*90.37	0.0091	0.0037
T144_004	22	80	31	124	88.73	0.0091	0.0071	*92.42	0.0278	0.0083
T144_008	31	160	31	248	92.60	0.0490	0.0202	*95.41	0.0873	0.0198
T144_016	44	320	31	496	92.42	0.2110	0.0300	*94.39	0.2399	0.0375
T144_032	62	640	31	992	95.41	0.9061	0.0814	*96.87	0.7115	0.1008
T144_064	88	1280	31	1984	96.44	3.8920	0.1909	*97.50	2.1135	0.2169
T144_128	124	2560	31	3968	96.87	16.8501	0.3864	*97.50	5.9863	0.4793
T144_256	176	5120	31	7936	98.01	73.0135	0.8748	*98.59	17.3983	1.0518
T144_512	249	10240	31	15872	98.38	328.4476	2.0470	*98.76	49.4838	2.3441

Figure 5.8 shows two example layouts obtained by the bottom-left and the best-fit algorithms. These are the layouts obtained for the instance named T144_004. The height obtained by the bottom-left algorithm is 25 and that obtained by the best-fit algorithm is 24. The occupation rates of these layouts are 88.73% and 92.42%, respectively, as reported in Table 5.11.

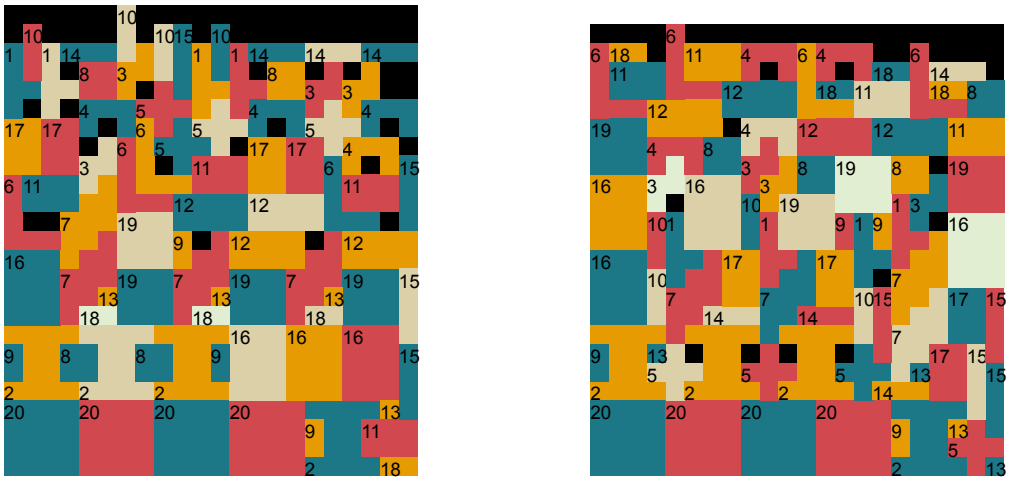


Figure 5.8: Layouts obtained for T144_004 by the two algorithms (left: bottom-left algorithm, right: best-fit algorithm)

For analyzing how performance changes for the cases when m is small relative to M and m is large relative to M, we tested our best-fit algorithm with the efficient implementations in Chapter 5 on instances named "HHIY" that were generated randomly. Theses instances are also publicly available and can be downloaded from http://www.co.cm.is.nagoya-u.ac.jp/~yagiura/rectilinear/.

Each rectilinear block was generated by combing up to six rectangles, and we generated four instances with 10, 100, 1000 and 10,000 distinct shapes. We then generated other instances by copying the shapes in these instances, e.g., the instance with 1000 blocks (i.e., $n = 1000$) with 100 distinct shapes was generated by making 10 copies of each shape in the instance with 100 distinct shapes. Table 5.12 shows the information of m and M for each instance of HHIY. The computational results of these instances are shown in Table 5.13. Each cell in these tables contains the result obtained for the instance that has n rectilinear blocks and t distinct shapes for the values of n and t in the corresponding row and column (e.g., the data in the cell in the row of "$n = 1000$" and column of "100 Shapes" is the result of the instance generated by making 10 copies of each shape in the instance with 100 distinct shapes). The column of "Occup." shows the occupation rate

in %. All running times are shown in seconds.

We can observe from the tables that, for every value of n, the running time increases almost linearly with the number of distinct shapes of these instances. Even for the instance with 10,000 distinct shapes, our algorithm runs in about 5798 seconds. According to the results, when the values of M are similar, the computation time is almost proportional to m.

Table 5.12: Information of HHIY

	10 Shapes		100 Shapes		1000 Shapes		10000 Shapes	
n	m	M	m	M	m	M	m	M
10	34	34	–	–	–	–	–	–
100	34	340	345	345	–	–	–	–
1000	34	3400	345	3450	3546	3546	–	–
10000	34	34000	345	34500	3546	35460	35014	35014

Table 5.13: Computational results for HHIY

	10 Shapes		100 Shapes		1000 Shapes		10000 Shapes	
n	Occup.	Time	Occup.	Time	Occup.	Time	Occup.	Time
10	69.23	0.0031	–	–	–	–	–	–
100	82.65	0.0330	81.29	0.4125	–	–	–	–
1000	84.11	0.3998	86.90	4.6103	94.20	53.8471	–	–
10000	85.14	4.6979	88.08	52.4316	95.42	562.6420	95.21	5798.3214

5.7 Conclusion

In this chapter, we proposed more efficient implementations of the bottom-left and the best-fit algorithms for the rectilinear block packing problem than those in Chapter 4, with which the bottom-left algorithm requires $O(M^2 \log M)$ time and the best-fit algorithm requires $O(nmM \log M)$ time.

We designed sophisticated data structures that dynamically keep the information so that the BL position of each item can be found in sub-linear amortized time. We then analyzed the time complexities of the two construction algorithms and showed that both algorithms run in $O(mM \log M)$ time.

We performed a series of experiments on a set of instances that are generated from nine benchmark instances to analyze the performance of our algorithms from both sides of occupation rate and running time. The computational results show that the proposed algo-

rithms are especially efficient for large instances with repeated shapes. Even for instances with more than 10,000 rectilinear blocks with up to 60 distinct shapes, the proposed algorithms run in less than 12 seconds. The occupation rate of the packing layouts obtained by our algorithms often reached higher than 95% for large-scale instances.

We compared the running time of our algorithms with the implementation explained in Chapter 4 and those with implementation proposed in this chapter. The computational results show that our efficient implementations significantly reduced the computation time of the bottom-left and the best-fit algorithms.

We also created large-scale instances with up to 10,000 distinct shapes to test our efficient algorithms and show that the running time increases almost linearly with the number of distinct shapes. For instances with 10,000 distinct shapes, our algorithms run in less than 6000 seconds.

In the next chapter, we further analyze the quality of packing layouts obtained by the bottom-left and the best-fit algorithm and propose a new construction heuristic.

Chapter 6

Partition-based Heuristic Algorithm for the Rectilinear Block Packing

In this chapter, we first analyze the strength and weakness of the bottom-left and the best-fit algorithms from the viewpoint of the quality of the packing results in Section 6.1. We summarize the reasons why the best-fit algorithm outperforms the bottom-left algorithm for many instances and situations when the bottom-left algorithm performs better for some kinds of instances.

Based on these observations, in Section 6.2, we propose a new construction heuristic algorithm called the *partition-based best-fit heuristic* (abbreviated as *PBF*) as a bridge between the bottom-left and the best-fit algorithms. The basic idea of the PBF algorithm is that all the items to be packed are partitioned into groups, and then items are packed into the container in a group-by-group manner. The best-fit algorithm is taken as the internal tactics to pack items of each group.

We then in Section 6.3 show that the PBF algorithm runs in the same time complexity using similar implementations explained in Chapter 5 of the bottom-left and the best-fit algorithms (these two algorithms have the same time complexity). We also give some effective rules to partition items into groups in Section 6.4. We finally perform a series of experiments on instances that were generated from nine benchmark instances. The computational results are addressed in Section 6.5.

6.1 Analysis of the performance of the bottom-left and the best-fit algorithms

In this section, we analyze the performance of the bottom-left and the best-fit algorithms according to the experimental results addressed in Chapter 5.

We observed that the best-fit algorithm performed better with respect to the occupation rate for many of the instances we tested. However, there are also a non-negligible number of instances for which the opposite holds. Observing and analyzing the packing layouts obtained by these two algorithms, we summarize the reason why the best-fit algorithm performs better for many of the instances we tested and the situations when the bottom-left algorithm performs better.

The reason why the best-fit algorithm performs better for many instances is that whenever the best-fit algorithm packs an item into the container, it tries all the remaining items relative to the current layout, and chooses the one that can be placed at the lowest position. As a result, an item that fits well with the surrounding layout tends to be chosen, which means that redundant space around the new item is usually small. On the contrary, the bottom-left algorithm may not choose a proper item that fits well with the current layout, because the next item to place is always fixed a priori (by the given order of items). Figure 6.1 shows an example when the best-fit algorithm performs better. The left layout of Figure 6.1 is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height obtained by the bottom-left algorithm is 40 and that obtained by the best-fit algorithm is 37.

Figure 6.1: An example when the best-fit algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

However, for some cases, e.g., when there are items whose areas are small but the bounding areas are large, and there are also items whose bounding areas are small, the bottom-left algorithm often performs better. Note that if the area of an item is small but its bounding area is large, it has large blank space inside (i.e., if it is put into its bounding box, large blank space remains in the bounding box in which small items can fit).

From practical experience it is known that the bottom-left algorithm tends to perform well when the given order is the order of decreasing sizes of items, where various measures for sizes can be considered such as the areas of items, their bounding areas, widths or heights. Let us consider the case where the algorithm packs items in the decreasing order of bounding area.

At the beginning of the packing process, the bottom-left algorithm packs relatively bigger items (with greater bounding area) into the container, and some of them have large blank space inside. At this moment, large spaces are often made between such large items. Later, when relatively smaller ones come, they tend to be packed into the blank space between the packed ones. This means that the placement of small ones does not have much influence on the final value of height H of the container, and H is mainly decided by the layout of bigger ones.

Conversely, because the BL positions of relatively smaller items are often lower than those of bigger ones, the best-fit algorithm tends to pack smaller ones first, and leave relatively bigger ones behind. In the end of the processing of the best-fit algorithm, the remaining bigger items have no choice but to place on the top of smaller ones, and the blank space between these bigger items have significantly negative effect on the final occupation rate. Figure 6.2 shows an example of this case. The left layout of Figure 6.2 is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height of the left one is 3960 and that of the right one is 4283.

For the same instance used in Figure 6.2, Figure 6.3 shows the layouts when the first half of the items are packed into the container. The left layout is obtained by the bottom-left algorithm, and the right one is obtained by the best-fit algorithm. The height of the left one is 3960 and that of the right one is 1880. At this moment, the height of the container obtained by the bottom-left algorithm is already the same as that of its final layout. This suggests that the remaining half of items have no effect on the final height of the container. On the contrary, many small items have already been packed by the best-fit algorithm, and by comparing the layouts on the right of Figures 6.2 and 6.3, we can observe that most of the remaining items are large.

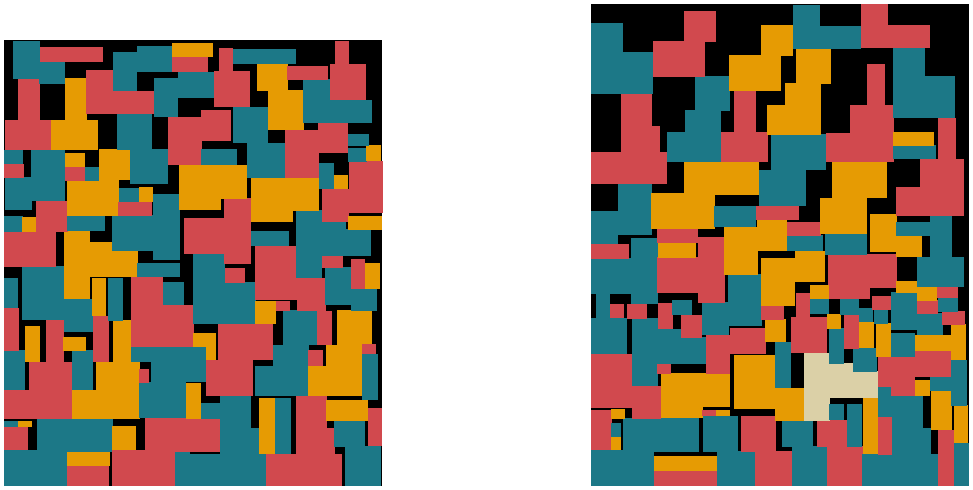


Figure 6.2: An example when the bottom-left algorithm performs better (left: the bottom-left algorithm, right: the best-fit algorithm)

6.2 Partition-based best-fit algorithm

In this section, we propose a new construction heuristic algorithm, the partition-based best-fit (PBF) algorithm, for the rectilinear block packing problem.

Considering the observation in Section 6.1, the idea of simply choosing either the best-fit algorithm or the bottom-left algorithm according to the property of instances comes naturally. Another simple idea would be just to run both algorithms and then choose the better layout. However, note the fact that the best-fit algorithm performs excellent in many cases when the sizes of items are similar. Hence, we propose a new construction heuristic algorithm, which uses the best-fit algorithm as its core part, but alleviates the drawback of the best-fit algorithm. The main idea is to divide the given rectilinear blocks into groups and then pack the items into the container in a group-by-group manner. We utilize the best-fit algorithm to pack the items in each group.

Intuitively, we would like to divide the items into groups so that the sizes of items in the same group are similar. There will be many rules to achieve this, regarding how to measure the sizes, how to divide items into groups and so forth. The rules we consider for measuring sizes and for dividing items into groups will be explained in Section 6.4.

Because there are many possible rules to divide the items into groups and to give priority among the items, we explain the framework of the PBF algorithm assuming that we are given a partition and priority among items. Then the PBF algorithm is generally explained as follows: We are given a set of n rectilinear items $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, which is divided into K groups $\mathcal{B} = \{B_1, B_2, \ldots, B_K\}$, and the priority among the items. The

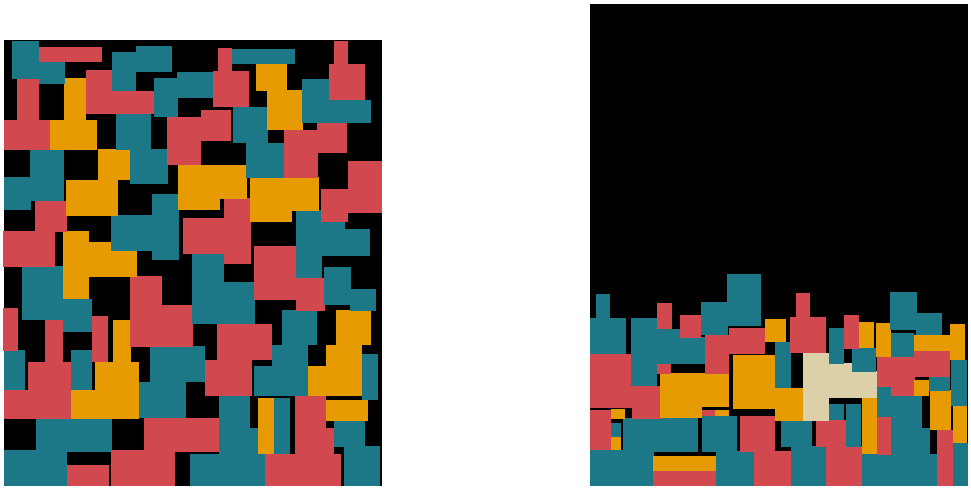


Figure 6.3: Layouts when the algorithms pack half of the items (left: the bottom-left algorithm, right: the best-fit algorithm)

PBF algorithm packs all of the groups one by one according to the given order, where each group is packed by the best-fit algorithm relative to the current layout (i.e., the layout at the time just before it is placed). The PBF algorithm is formally described as Algorithm 16.

Algorithm 16 PBF algorithm

Input: K groups of rectilinear blocks $B_1, B_2, \ldots, B_K$ and a container.

Output: A packing layout.

1: Set $k := 1$.

2: For the current layout, call the best-fit algorithm to pack all the items in group B_k into the container.

3: If $k = K$, output the current layout and stop; otherwise, set $k := k + 1$, and then return to Step 2.

Note that, if $K = 1$, the PBF algorithm performs the same as the best-fit algorithm. If $K = n$, the processing of PBF algorithm is the same as the bottom-left algorithm. In this sense, the PBF algorithm generalizes the bottom-left and the best-fit algorithms, bridging the gap between them.

6.3 Time complexity

In this section, we analyze the time complexity of the PBF algorithm. As explained in Section 5.5, m_i is the number of rectangles that represent a rectilinear block R_i and m_j^{T}

is the number of rectangles that represent shape T_j. Recall also that $M = \sum_{i=1}^{n} m_i$ and $m = \sum_{j=1}^{t} m_j^{\mathrm{T}}$.

We can implement the PBF algorithm so that is runs in the same time complexity as the best-fit algorithm, which equals $O(mM \log M)$ time, by slightly modifying the efficient implementations proposed in Chapter.

Below we briefly explain the basic idea of the efficient implementation of the best-fit algorithm. We utilized the technique of no-fit polygon (*NFP*) to check overlaps among items and to compute the BL positions of items. The NFP of an item R_j relative to R_i has the following property: the reference point of R_j is contained in the NFP if and only if R_j overlaps with R_i. When the algorithm computes the BL position of an item R_j, it uses the NFPs of R_j relative to the items in the container, and such NFPs are placed at the positions where the corresponding items are placed. We call such a layout of NFPs an NFP layout for R_j. One of the advantages of such a layout is that the problem of finding the BL position of R_j reduces to the problem of finding the leftmost position among the lowest positions that are not contained in any of the NFPs in the NFP layout. Note that if the shape of two items R_j and $R_{j'}$ are the same, their NFP layouts are the same. Hence, we only need to keep $t = |\mathcal{T}|$ NFP layouts, each for a distinct shape in $\mathcal{T}$, to compute BL positions for the remaining items.

A common feature of construction heuristics is that once an item is packed into the container, its position is fixed and will not change. This indicates that it is not necessary to compute NFP layouts from scratch in each iteration of the construction heuristics. The basic idea is to dynamically keep the NFP layout with respect to the current packing layout for each shape in $\mathcal{T}$ during the packing process. Whenever an item is to be placed into the container, the algorithm computes the BL position of every shape T_j by using the NFP layout for T_j. It then chooses an item R_i to place in this iteration (i.e., the item whose BL position is the leftmost among the lowest) and place it at its BL position. The algorithm then updates the NFP layout of every shape T_j in $\mathcal{T}$, adding to the NFP layout the NFP of T_j relative to R_i.

We analyzed in Section 5.5 the time complexity of the best-fit algorithm and summarized the result in Theorem 5.5.1 that throughout the entire computation of the best-fit algorithm, the total running time for computing BL positions of shape T_j is $O(m_j^{\mathrm{T}} M \log M)$, including the time to update the NFP layout of T_j. The total computation time of this part for all shapes is therefore $\sum_{j=1}^{t} O(m_j^{\mathrm{T}} M \log M) = O(mM \log M)$. This dominates the running time of the other parts of the algorithm. As a consequence, the best-fit algorithm runs in $O(mM \log M)$ time.

The point is that, with this implementation, the BL positions of all the remaining items are available in every iteration. We can therefore implement the PBF algorithm

similarly, just by considering in each iteration, the items belonging to the current group to be packed in this iteration, instead of considering all the remaining items in choosing the next item to place. Such a modification does not increase the execution time from that of the best-fit algorithm. Hence, with an implementation similar to the best-fit algorithm, the PBF algorithm runs in $O(mM \log M)$ time.

Theorem 6.3.1. *Assuming that M is the number of rectangles that represent all rectilinear blocks and m is the number of rectangles that represent all distinct shapes of rectilinear blocks, the PBF algorithm runs in $O(mM \log M)$ time.*

6.4 Partition rules

In this section, we explain some rules that are utilized to partition the given items into groups. Our rules are based only on the shapes and sizes of items, and for this reason, any two rectilinear blocks with the same shape and size are always in the same group. Hence the partition of items can be defined by a partition of shapes in $\mathcal{T}$.

We consider a framework in which the PBF algorithm (Algorithm 1) is repeatedly called with a series of partitions. Algorithm 1 is first applied to the partition consisting of only one block. Then in each iteration, Algorithm 1 is applied to a partition generated by dividing a group in the partition for the previous iteration into two (i.e., in each iteration, the number of groups in a partition increases by one).

We explain the rules of how to partition a group into two in Section 6.4.1, and the rules to choose a group to be divided in Section 6.4.2. We test the PBF algorithm on a series of instances based on these rules and the computational results will be addressed in Section 6.5.

6.4.1 Rules to partition a group

Recalling the analysis of the performance of the bottom-left and the best-fit algorithm discussed in Section 6.1, the essential purpose of our rules is to pack at the early stage, the items that are "difficult" for the algorithm to pack or that have "negative" impact on the final occupation rate. We propose two types of rules, the *static* rules and the *adaptive* rules, to measure such "difficulty" or "negative impact" of the items.

Static rules

The static rules divide a given group in two parts according to the information of properties of the items themselves. We consider various rules to partition one group. Considering the analysis that is explained in Section 6.1, we propose *size-based* rules in which the sizes are

measured by the values of the areas, bounding areas, widths or heights of the items. First we sort the items in the given group in decreasing order of size with respect to a criterion (e.g., area) and then divide the group into two at the place between two adjacent items in the sorted list where the gap with respect to the size criterion is the largest. For example, assume that there is a group with five shapes of rectilinear blocks $\mathcal{G} = \{T_1, T_2, T_3, T_4, T_5\}$ whose areas are "$12, 2, 10, 5, 11$," respectively. If we use the size-based rule with area for the criterion, we sort the shapes in $\mathcal{G}$ in the decreasing order of areas and obtain the sequence "T_1, T_5, T_3, T_4, T_2" and the corresponding sequence of areas "$12, 11, 10, 5, 2$." The gap between the first and second items is $12 - 11 = 1$, that of the second and third is $11 - 10 = 1$, and so forth. We divide the group $\mathcal{G}$ between the third and fourth items where the biggest gap of 5 is achieved, obtaining two new groups $\mathcal{G}_1 = \{T_1, T_3, T_5\}$ and $\mathcal{G}_2 = \{T_2, T_4\}$.

We also consider another static rule named *inclusion-based* rule. For two shapes T_i and $T_j \in \tilde{\mathcal{T}}$ where $\tilde{\mathcal{T}}$ is the current group to partition, $T_i \preceq T_j$ signifies that T_i and T_j can be packed into the bounding box of T_j without overlap. We divide the items in the given group into two parts LARGE and SMALL so that for every item T_i in SMALL, there exists at least one item T_j in the given group that satisfies $T_i \preceq T_j$. This rule can be formally described as follows:

$$\text{SMALL} = \{T_i \in \tilde{\mathcal{T}} \mid \exists T_j \in \tilde{\mathcal{T}}, T_i \preceq T_j\},$$
$$\text{LARGE} = \tilde{\mathcal{T}} \setminus \text{SMALL}.$$

Adaptive rules

The adaptive rules utilize the information of the BL positions of the rectilinear blocks in the given group relative to a packing layout obtained in the current iteration (i.e., the latest call to Algorithm 1), in order to generate a partition for the next iteration. In computing the BL positions that are utilized for this purpose, we consider two strategies and call the resulting rules *BL-midway* and *BL-final*.

In the rule of BL-midway, for each group B_k, we recompute the BL position and store it for every shape T_i in B_k relative to the packing layout at the time immediately after all the rectilinear blocks in group B_k have been packed into the container. (Note that at the time such BL positions are recomputed, all items with shape T_i for every shape T_i in B_k have already been packed, and such recomputed BL positions will not be used to actually place rectilinear blocks, but just to generate a partition for the next iteration. Note also that we do not have to actually recompute BL positions, because they are automatically obtained whenever an item is placed and the NFP layouts are updated, and hence the recomputation of BL positions will not increase the time complexity.)

In the rule of BL-final, we recompute the BL position and store it for every shape T_i relative to the packing layout after packing all the rectilinear blocks into the container (i.e., we store the BL position of every shape relative to the final packing layout). Then, we sort the shapes in the group to divide in the decreasing order of the y-coordinates of their recomputed BL positions. We then divide the group into two at the place between two adjacent items in the sorted order where the difference of y-coordinates is largest.

6.4.2 Rules to choose a group to divide

In this section, we give some rules to choose a group to divide next.

Considering the groups that are obtained by the inclusion-based rule, only the group of SMALL could be divided. Indeed, with this rule, the LARGE group cannot be divided any more for the following reason. For every pair of shapes T_i and T_j in LARGE, $T_i \preceq T_j$ cannot hold because otherwise, $T_i \preceq T_j$ and $T_j \in \text{LARGE} \subseteq \tilde{\mathcal{T}}$ would imply that T_i must have been in SMALL, which is a contradiction. Hence, we always choose SMALL for the group to divide next. This partition process terminates when there is less than two shapes in SMALL.

For the other rules we explained in Section 6.4.1, we propose four rules to choose the group to divide next time. When these rules are used, partition process terminates when the chosen group contains a unique shape.

We first give three simple rules labeled *FirstGrp*, *LastGrp* and *LargeGrp* as follows. Below we assume that groups are sorted according to the order of items with respect to the adopted criterion. For example, assuming that the decreasing order of area is considered, when a group is divided into two, the group containing items with larger area is listed first, and the two new groups are placed at the place of the original group (before the division is applied) in the list of groups.

- In the FirstGrp rule, we always choose the first group that contains more than one shape.
- In the LastGrp rule, we always choose the last group that contains more than one shape.
- In the LargeGrp rule, we choose the group with the largest size (i.e., the group that contains the largest number of shapes). If there is more than one group with the largest size, we break the tie by choosing the first one among them.

Considering the analysis of the performance of the bottom-left and the best-fit algorithm, one of our intentions of partitioning the rectilinear blocks into groups is to make the best-fit algorithm perform well in packing each group into the container. To achieve

this, it is preferable that the sizes of items in the same group are similar. Hence, whenever we choose a group to divide, it is natural to choose a group that contains shapes with large differences in their sizes. Recall that in the rules of Section 6.4.1 to partition one group into two (except for the inclusion-based rule), the group is always divided at a place between two adjacent items in the sorted order where the gap with respect to the considered criterion is the largest. The value of this gap will have correlation with the differences among the sizes of items in one group. As a consequence, we consider another rule named *BigGapGrp* to choose the group to divide as follows.

- In the BigGapGrp rule, we choose the group with the biggest gap, where the gap is the one used by an adopted partition rule. If there exist ties, the group with the largest size is chosen.

As explained at the beginning of this section, any two rectilinear blocks with the same shape are in the same group. This indicates that the PBF algorithm terminates after at most $t-1$ iterations, if we partition one group into two in each iteration. Note that, when we begin with one group, the processing of the PBF algorithm is the same as the best-fit algorithm, and after $t-1$ iterations, when items are divided into t groups of distinct shapes, it becomes the same as the bottom-left algorithm.

6.5 Computational results

The PBF algorithm proposed in this paper was also implemented in the C programming language and run on a Mac PC with a 2.3 GHz Intel Core i5 processor and 4 GB memory.

The performance of the PBF algorithm has been tested on a series of instances, which are used to test the efficient implementations proposed in Chapter 5. The information of the benchmark instances and those in C-class are addressed in Table 5.1 and 5.2 in Section 5.6. The instances in C-class were generated from the benchmark instances by copying every shape.

For analyzing the performance of the algorithms for the case when the sizes of rectilinear blocks are quite different, we also test the bottom-left, the best-fit and the PBF algorithms on the another set of instances named *E-class* that are generated by adding enlarged shapes of the instances in C-class. For convenience, we again address the information of instances in C-class in Table 6.1. Each rectilinear block in an instance of E-class is generated by enlarging the size of one item in the corresponding instance in Table 6.1 by one, two, four or eight times. The information of the instances in E-class is addressed in Table 6.2. The width W of the container is set to $W = \left\lceil \sqrt{\sum_{i=1}^{n} A(R_i)} \right\rceil$ according to the total sum of areas of items. The column of "#inst" reports the number of

instances in each class. All of these instances are available at http://www.co.cm.is.nagoya-u.ac.jp/~yagiura/rectilinear/.

Table 6.1: Information of C-class instances

Name	#inst	t	n	m	M	W
C-ami49L21	10	28	28–14336	49	49–25088	5936–134337
C-ami49LT21	10	27	27–13924	49	49–25088	5953–134714
C-TMCNCGSRC	9	51	51–13056	76	76–19456	2376–38018
C-B10	12	9	9–18432	14	14–28672	9–438
C-B30	10	29	29–14848	59	59–30208	29–676
C-T19	11	19	19–19456	42	42–43008	18–601
C-T40	10	32	32–16384	42	42–21504	301–6818
C-T64	11	15	15–15360	33	33–33792	7–249
C-T144	10	20	20–10204	31	31–15872	11–249

Table 6.2: Information of E-class instances that are generated by adding enlarged copies of shapes of different sizes

Name	#inst	t	n	m	M	W
E-ami49L21	8	112	112–14366	196	196–25088	54735–619265
E-ami49LT21	8	108	108–13824	196	196–25088	54889–621004
E-TMCNCGSRC	7	204	204–13056	305	305–19456	21906–175255
E-B10	10	36	36–18432	56	56–28672	89–2022
E-B30	8	116	116–14848	236	236–30208	275–3118
E-T19	9	76	76–19456	168	168–43008	173–2771
E-T40	8	128	128–16384	168	168–21504	2778–31430
E-T64	9	60	60–15360	132	132–33792	72–1152
E-T144	8	80	80–10240	124	124–15872	101–1152

As for the order of items for the bottom-left algorithm and the priority among items for the best-fit algorithm, we tested the decreasing order of height, width, area, and bounding area. The computational results of the decreasing order of area is slightly better than the results obtained by other orders. Hence, we report those results of the decreasing order of area. We define the *BestOccup* as the best occupation rate of an instance found when the processing terminates. The processing of the PBF algorithm begins with one group and is repeated until no further partition is possible for the chosen group. *AvgOccup* is the average value of BestOccup over all instances in each class.

The occupation ratios obtained by the bottom-left, best-fit and the PBF algorithms are shown in Table 6.3 and 6.4. The average for all instances in each set of instances is reported. The columns of *BL* and *BF* show the avgOccup in percent obtained by the bottom-left and the best-fit algorithms. The column of Incl-based is the avgOccup in percent obtained by the PBF algorithm in which we utilize the Inclusion-based rule and the priority among items is set to the decreasing order of area. For each set of instances, the best results among the seven algorithms are marked by '$*$'.

Table 6.3: Comparison of computational results with a fixed rule to choose the group to divide next

Instance	BL	BF	Incl-based	FirstGrp	LastGrp	LargeGrp	BigGapGrp
C-ami49L21	88.10	87.32	88.51	*90.24	89.97	89.99	90.00
C-ami49LT21	87.83	87.10	88.79	90.26	89.82	90.03	*90.26
C-TMCNCGSRC	86.92	81.71	88.93	*90.18	90.05	90.17	90.00
C-B10	87.66	90.32	90.59	*92.06	*92.06	91.74	91.74
C-B30	84.52	88.14	88.31	*89.42	89.25	89.18	89.19
C-T19	73.79	76.24	76.61	*78.55	78.35	78.35	78.41
C-T40	94.09	91.24	94.33	*96.62	96.30	96.30	96.54
C-T64	86.99	92.92	92.92	*93.50	*93.50	*93.50	*93.50
C-T144	91.11	93.39	96.40	*96.90	96.60	96.60	96.45
E-ami49L21	94.71	89.51	94.86	95.65	*95.70	95.69	95.69
E-ami49LT21	94.72	87.66	94.63	95.46	95.62	*95.70	95.62
E-TMCNCGSRC	92.89	83.55	93.18	93.58	93.94	*94.08	93.88
E-B10	93.93	87.92	94.75	96.10	96.55	*96.71	96.69
E-B30	94.58	89.20	95.27	95.55	95.59	95.63	*95.90
E-T19	86.95	84.19	88.42	88.94	*89.69	89.58	89.57
E-T40	98.01	90.63	98.33	98.80	98.88	*98.94	98.90
E-T64	95.33	92.87	96.42	97.19	97.49	97.48	*97.54
E-T144	97.83	96.71	98.32	98.74	98.73	*98.95	98.73

Table 6.4: Comparison of computational results with a fixed rule to divide a group

			Static					Adaptive	
Instance	BL	BF	Incl-based	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	88.51	89.58	89.75	89.28	89.50	90.12	*90.23
C-ami49LT21	87.83	87.10	88.79	89.76	89.99	88.92	89.47	90.13	*90.30
C-TMCNCGSRC	86.92	81.71	88.93	89.93	90.08	87.67	88.26	*90.15	90.07
C-B10	87.66	90.32	90.59	90.36	91.28	90.25	91.15	*91.68	*91.68
C-B30	84.52	88.14	88.31	88.30	88.81	88.63	88.42	*89.16	88.98
C-T19	73.79	76.24	76.61	76.63	77.16	77.66	75.67	*77.87	77.49
C-T40	94.09	91.24	94.33	96.16	96.13	94.88	*96.30	95.29	95.21
C-T64	86.99	92.92	92.92	*93.43	92.92	93.24	92.87	92.92	92.92
C-T144	91.11	93.39	*96.40	96.30	96.14	95.21	95.30	95.72	95.01
E-ami49L21	94.71	89.51	94.86	95.34	95.51	95.06	95.32	*95.72	95.70
E-ami49LT21	94.72	87.66	94.63	95.29	95.39	95.00	95.30	*95.65	95.62
E-TMCNCGSRC	92.89	83.55	93.18	*94.09	93.83	92.26	92.95	93.84	93.60
E-B10	93.93	87.92	94.75	95.58	95.48	95.16	*96.61	95.98	95.93
E-B30	94.58	89.20	95.27	95.39	95.44	94.89	94.91	*95.61	95.57
E-T19	86.95	84.19	88.42	87.64	89.22	87.49	88.86	89.23	*89.36
E-T40	98.01	90.63	98.33	98.90	98.74	97.64	*98.92	98.71	98.70
E-T64	95.33	92.87	96.42	96.91	96.53	96.42	*97.41	97.31	96.98
E-T144	97.83	96.71	98.32	98.72	98.64	98.31	*98.82	98.30	97.87

In Table 6.3, we address the computational results obtained by the PBF algorithm when we use the rules of FirstGrp, LastGrp, LargeGrp and BigGapGrp for choosing a group to divide next. The columns of FirstGrp, LastGrp, LargeGrp and BigGapGrp are the avgOccup in percent of each class obtained by the PBF algorithm when utilizing the rules of FirstGrp, LastGrp, LargeGrp and BigGapGrp to choose a group to divide. For each instance set (i.e., for each row), the table shows the average of the best result obtained for each instance by multiple calls to the PBF algorithm with different rules in Section 6.4.1 to divide one group (except for the Inclusion-based rule). For example, the value of "90.24%" in the cell of row C-ami49L21 and column FirstGrp is the average of the values of BestOccup for all the instances in class C-ami49L21, where the BestOccup of every instance is the best occupation rate when we try all of the Size-based and the adaptive rules to partition the first group.

We also summarize the results obtained when using the different rules to partition one group in two. We report the computational results obtained when we use the rules of Size-based and the adaptive rules to partition one group in Table 6.4. The columns of *BndArea* (bounding area), *Area*, *Width*, *Height* in *Static* and *BL-midway* and *BL-final* in *Adaptive*

are the avgOccup in percent of the BestOccup that are obtained by the PBF algorithm when we fix these rules to partition one group and examine all the rules introduced in Section 6.4.2 for choosing the group to divide next. For example, the value of "89.58%" in the cell of row C-ami49L21 and column BndArea is the average of the values of BestOccup for all the instances in class C-ami49L21, where the BestOccup of every instance is the best occupation rate when we try all of rules to choose one group and divide it according to the values of the corresponding criterion from bounding areas of the items. We use the decreasing order of area as the priority among the items when adaptive rules are utilized. If the Size-based rules are used to partition one group, the priority among the items is decided by the decreasing order of the values of the bounding areas, areas, widths and heights of the items.

The computational results show that the PBF algorithm improves the occupation rate significantly compared with the bottom-left and the best-fit algorithms, and the improvement is more remarkable for the case where the sizes among the rectilinear blocks are much different (i.e., the instances in E-classes).

The results in Table 6.3 show that the PBF algorithm performs best for the instances whose sizes are not much different (i.e., instances in C-classes) when the algorithm chooses the group to divide with the FirstGrp rule. If there are large differences in the sizes of the rectilinear blocks (i.e., instances in E-classes), the PBF algorithm performs better for most of such instances when the algorithm chooses the group to divide with the LargeGrp rule. Note that even for the cases where the PBF algorithm did not obtain the best results with the above rules, the difference in the occupation rate is less than 0.77% between the results obtained by the best and the worst rules among the four rules (i.e., FirstGrp, LastGrp, LargeGrp and BigGapGrp). This suggests that the performance of the PBF algorithm is robust against the rules of choosing one group to divide.

The big difference among the results in Table 6.4 indicates that the rules to partition one group into two are very important for the PBF algorithm. The PBF algorithm performs better for most of the instances when using the adaptive rule BL-midway or the static rule Height. Even for the classes where the occupation rates obtained by these two rules are not the best, the difference between the best of these two rules and the best among all rules is less than 0.68%.

Figure 6.4 shows three example layouts obtained by the bottom-left, the best-fit and the PBF algorithms. These are layouts obtained for the instance named E-TMCNCGSRC_001. The occupation rate obtained by the bottom-left algorithm is 90.59%, that obtained by the best-fit algorithm is 79.51%, and that obtained by the PBF algorithm is 92.45%. The running time spent for these layouts are reported in Table 6.7.

We report the running time spent by the bottom-left, the best-fit and the PBF algo-

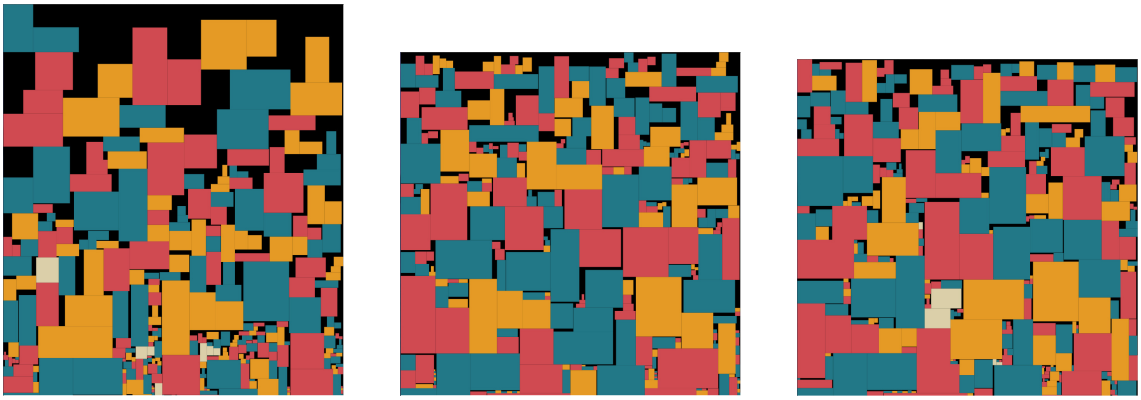


Figure 6.4: Layouts obtained for E-TMCNCGSRC_001 by the three algorithms (left: the best-fit algorithm, middle: the bottom-left algorithm, right: the PBF algorithm)

rithms in Table 6.5 to 6.7. All the running times are shown in seconds. Table 6.5 shows the running time spent for the instance with the largest size in every class. We also report the running times for instances with various sizes. For this purpose, we choose TMCNCGRSC because its computation times in Table 6.5 are the longest for both of the categories of the C- and E-classes. Table 6.6 and 6.7 address the running times for instances in classes C-TMCNCGSRC and E-TMCNCGSRC, respectively. The columns of *BL-Time* and *BF-Time* show the running times of the bottom-left and the best-fit algorithms. The running time of the PBF algorithm when using the Inclusion-based rule to partition the SMALL group are addressed in column *Incl-time*. For other pairs of rules to choose and partition a group (e.g., the combination of LargeGrp and BL-final), the PBF algorithm is repeated $t-1$ times, and we consider the total computation time for the $t-1$ iterations. The combination of such rules do not have a large influence on the computation time except that the PBF algorithm with the adaptive rule BL-midway and BL-final tend to spend slightly more computation time. For this reason, we do not report the results obtained by all pairs of rules but just report the running time of the PBF algorithm with the combination of LargeGrp and BL-final rules in column *Largest-Final*.

We address in Table 6.8 to 6.11 the details of the computational results for each class of instances by the PBF algorithm when it uses one of the rules from FirstGrp, LastGrp, LargeGrp and BigGapGrp to choose a group to divide. The BestOccup of every instance is the best occupation rate obtained by the PBF algorithm with a specified pair of rules for choosing and partitioning groups. As explained in Section 6.5, the avgOccup is defined as the average value of BestOccup over all instances in one class. The columns of *BndArea*, *Area*, *Width* and *Height* are the avgOccup in percent obtained by the PBF algorithm with the Size-based rules based on the values of the bounding areas, areas, widths and heights

Table 6.5: Computation time for the largest instances in each class

Instance	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
C-ami49L21_512	28	14336	49	25088	5.12	6.52	18.26	147.64
C-ami49LT21_512	27	13924	49	25088	5.51	7.00	22.17	166.96
C-TMCNCGSRC_256	51	13056	76	19456	6.23	8.91	45.64	380.30
C-B10_2048	9	18432	14	28672	1.52	1.82	3.64	12.56
C-B30_512	29	14848	59	30208	8.25	9.13	31.45	247.60
C-T19_1024	19	19456	42	43008	7.91	8.42	41.46	147.35
C-T40_512	32	16384	42	21504	3.45	5.62	20.62	144.43
C-T64_1024	15	15360	33	33792	4.62	4.95	24.67	68.20
C-T144_512	20	10204	31	15872	2.05	2.62	18.43	45.80
E-ami49L21_128	112	14366	196	25088	21.08	29.02	230.56	2997.43
E-ami49LT21_128	108	13824	196	25088	19.67	24.61	158.74	2442.70
E-TMCNCGSRC_064	204	13056	305	19456	26.11	35.73	346.61	6609.45
E-B10_512	36	18432	56	28672	5.64	7.65	54.62	249.38
E-B30_128	116	14848	236	30208	28.47	38.62	367.72	3450.55
E-T19_256	76	19456	168	43008	27.97	36.52	400.45	2347.43
E-T40_128	128	16384	168	21504	15.38	21.90	130.73	2413.09
E-T64_256	60	15360	132	33792	15.44	21.07	257.26	1103.24
E-T144_128	80	10240	124	15872	7.00	9.09	110.62	628.38

Table 6.6: Computation time for instances in class C-TMCNCGSRC

Instance	W	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
C-TMCNCGSRC_001	2376	51	51	76	76	0.01	0.01	0.07	0.56
C-TMCNCGSRC_002	3360	51	102	76	152	0.02	0.03	0.15	1.35
C-TMCNCGSRC_004	4752	51	204	76	304	0.06	0.06	0.38	3.09
C-TMCNCGSRC_008	6720	51	408	76	608	0.11	0.14	0.82	6.35
C-TMCNCGSRC_016	9504	51	816	76	1216	0.26	0.36	2.00	16.24
C-TMCNCGSRC_032	13441	51	1632	76	2432	0.60	0.88	4.78	40.51
C-TMCNCGSRC_064	19009	51	3264	76	4864	1.34	1.86	9.72	90.77
C-TMCNCGSRC_128	26882	51	6528	76	9728	2.91	3.90	19.46	178.45
C-TMCNCGSRC_256	38018	51	13056	76	19456	6.23	8.91	45.64	380.30

Table 6.7: Computation time for instances in class E-TMCNCGSRC

Instance	W	t	n	m	M	BL-Time	BF-Time	Incl-Time	Largest-Final
E-TMCNCGSRC_001	21906	204	204	304	304	0.20	0.26	1.64	48.72
E-TMCNCGSRC_002	30981	204	408	304	608	0.46	0.68	3.74	113.68
E-TMCNCGSRC_004	43813	204	816	304	1216	1.03	1.59	8.56	253.87
E-TMCNCGSRC_008	61962	204	1632	304	2432	2.25	3.56	20.62	610.46
E-TMCNCGSRC_016	87627	204	3264	304	4864	5.51	8.67	43.73	1523.50
E-TMCNCGSRC_032	123924	204	6528	304	9728	12.31	18.80	90.63	3248.41
E-TMCNCGSRC_064	175255	204	13056	304	19456	26.11	35.73	346.61	6609.45

of items. The columns of BL-midway and BL-final are the avgOccup in percent obtained with the adaptive rules BL-midway and BL-final. We choose the decreasing order of area as the priority among the items when the adaptive rules are utilized. When the Size-based rules are used, the priority among the items is set to the same criterion as the one used to partition a group.

6.6 Conclusion

In this chapter, we proposed a new constructive heuristic algorithm, the partition-based best-fit (PBF) algorithm, for the rectilinear block packing problem. The PBF algorithm can be regarded as a bridge between the bottom-left and the best-fit algorithms and takes advantages of both of these two algorithms. We analyzed the time complexity of the proposed PBF algorithm and showed that it runs in $O(mM \log M)$ time with the efficient implementation explained in Chapter 5.

We also proposed some effective rules utilized in the PBF algorithm and performed a series of experiments on 168 instances that were generated from nine benchmark instances. The occupation rate of the packing layouts obtained by the proposed PBF algorithm was more than 93% on average for these instances. The computational results show that the improvement on the performance of the occupation rate obtained by the PBF algorithm is remarkable compared with the bottom-left and the best-fit algorithms, and the PBF algorithm is especially effective for instances with many different sizes of shapes.

Table 6.8: Computational results using the rule FirstGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.54	89.09	89.04	89.28	*89.61	89.56
C-ami49LT21	87.83	87.10	*89.67	88.99	88.68	89.38	89.48	89.52
C-TMCNCGSRC	86.92	81.71	89.08	89.59	87.56	87.61	89.49	*89.70
C-B10	87.66	90.32	90.36	91.28	89.95	90.82	91.28	*91.68
C-B30	84.52	88.14	88.17	88.50	88.63	88.42	*89.16	88.91
C-T19	73.79	76.24	76.60	77.16	77.66	75.62	*77.68	77.32
C-T40	94.09	91.24	95.89	95.95	94.47	*96.02	94.60	94.88
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	96.14	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.18	94.96	94.43	94.78	*95.51	95.42
E-ami49LT21	94.72	87.66	94.96	95.02	94.14	95.10	*95.40	95.07
E-TMCNCGSRC	92.89	83.55	*93.40	92.95	91.51	92.30	93.31	93.16
E-B10	93.93	87.92	94.72	94.42	94.94	94.30	*95.45	95.36
E-B30	94.58	89.20	94.64	95.04	94.55	93.83	*95.33	95.33
E-T19	86.95	84.19	87.38	*88.61	86.06	87.76	88.27	87.88
E-T40	98.01	90.63	*98.74	98.60	97.58	98.68	98.42	98.60
E-T64	95.33	92.87	95.52	95.93	95.04	95.73	*96.33	95.96
E-T144	97.83	96.71	*98.72	98.14	98.31	98.33	98.12	97.82

Table 6.9: Computational results using the rule LastGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.48	89.15	89.08	89.22	89.57	*89.68
C-ami49LT21	87.83	87.10	89.29	89.10	88.74	88.98	89.51	*89.57
C-TMCNCGSRC	86.92	81.71	*89.92	89.58	87.46	87.98	89.18	88.96
C-B10	87.66	90.32	90.36	91.28	90.25	91.15	*91.35	*91.35
C-B30	84.52	88.14	87.88	88.50	88.54	88.42	88.52	*88.58
C-T19	73.79	76.24	76.51	77.14	77.56	75.22	*77.68	77.15
C-T40	94.09	91.24	95.67	95.98	94.87	*96.20	94.70	94.69
C-T64	86.99	92.92	*93.43	92.92	93.24	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.31	95.21	95.30	95.24	94.40
E-ami49L21	94.71	89.51	95.27	*95.48	95.02	95.17	95.33	95.47
E-ami49LT21	94.72	87.66	94.91	95.29	95.00	95.18	*95.30	95.08
E-TMCNCGSRC	92.89	83.55	*93.88	93.78	92.15	92.76	93.48	93.35
E-B10	93.93	87.92	95.58	95.33	95.06	*96.33	95.76	95.64
E-B30	94.58	89.20	95.02	95.31	94.63	94.72	95.22	*95.37
E-T19	86.95	84.19	87.42	89.08	87.35	88.52	*89.09	88.55
E-T40	98.01	90.63	*98.77	98.63	97.61	98.76	97.81	98.08
E-T64	95.33	92.87	96.38	96.34	96.42	*97.34	96.45	96.41
E-T144	97.83	96.71	*98.72	98.00	98.14	98.58	98.15	97.81

Table 6.10: Computational results using the rule LargeGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	*89.48	89.04	89.07	89.37	89.45	89.40
C-ami49LT21	87.83	87.10	89.34	89.12	88.69	89.41	*89.75	89.62
C-TMCNCGSRC	86.92	81.71	*89.82	89.60	87.49	88.09	89.47	89.51
C-B10	87.66	90.32	90.36	91.28	90.25	90.82	90.96	*91.35
C-B30	84.52	88.14	87.88	88.50	88.54	88.42	*88.68	*88.68
C-T19	73.79	76.24	76.51	77.14	77.66	75.51	*77.68	77.15
C-T40	94.09	91.24	95.61	95.87	94.87	*96.20	94.84	94.71
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.31	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.34	95.43	95.02	95.18	95.42	*95.49
E-ami49LT21	94.72	87.66	95.07	95.24	94.79	95.26	95.47	*95.52
E-TMCNCGSRC	92.89	83.55	*94.08	93.77	92.15	92.63	93.55	93.31
E-B10	93.93	87.92	95.43	95.34	94.99	*96.51	95.77	95.44
E-B30	94.58	89.20	95.17	95.20	94.69	94.65	95.39	*95.42
E-T19	86.95	84.19	87.34	*89.21	86.90	88.30	89.08	88.96
E-T40	98.01	90.63	98.84	98.60	97.60	*98.92	98.08	98.11
E-T64	95.33	92.87	96.89	96.45	96.38	*97.34	96.37	96.41
E-T144	97.83	96.71	98.72	98.43	98.14	*98.82	98.10	97.80

Table 6.11: Computational results using the rule BigGapGrp to choose a group

			Size-based				Adaptive	
Instance	BL	BF	BndArea	Area	Width	Height	BL-midway	BL-final
C-ami49L21	88.10	87.32	89.47	89.44	88.98	89.22	89.59	*89.77
C-ami49LT21	87.83	87.10	89.57	89.72	88.79	89.30	89.69	*89.78
C-TMCNCGSRC	86.92	81.71	*89.50	88.61	87.42	88.03	89.49	89.40
C-B10	87.66	90.32	90.36	91.28	90.25	90.82	*91.35	*91.35
C-B30	84.52	88.14	88.30	*88.81	88.54	88.42	88.68	88.68
C-T19	73.79	76.24	76.51	76.85	77.48	75.55	*77.77	77.15
C-T40	94.09	91.24	96.06	94.90	94.49	*96.25	95.13	94.62
C-T64	86.99	92.92	*93.43	92.92	92.52	92.87	92.92	92.92
C-T144	91.11	93.39	*96.30	95.12	95.21	95.30	95.01	95.01
E-ami49L21	94.71	89.51	95.17	95.13	94.53	95.04	*95.60	95.43
E-ami49LT21	94.72	87.66	95.20	95.02	94.28	95.14	*95.53	95.44
E-TMCNCGSRC	92.89	83.55	93.50	93.11	91.81	92.79	*93.72	93.34
E-B10	93.93	87.92	95.45	94.39	94.99	*96.51	95.72	95.67
E-B30	94.58	89.20	95.33	95.24	94.78	94.82	*95.48	95.44
E-T19	86.95	84.19	87.37	88.71	86.85	88.25	88.88	*88.98
E-T40	98.01	90.63	98.74	98.64	97.62	*98.84	98.41	98.31
E-T64	95.33	92.87	96.69	96.45	96.38	*97.41	96.27	96.27
E-T144	97.83	96.71	*98.72	98.51	98.14	98.42	98.06	97.82

Chapter 7

Extension for Packing with Rotation

In this chapter, we explain how we can apply the efficient data structure in Chapter 5 to the rectilinear block packing problem with rotation.

Natural ways to generalize the bottom-left algorithm would be the following:

- The order of items and their orientations (i.e., the rotation angle) are given, and then the bottom-left algorithm for the case without rotation is applied.
- The order of items is given, and in each iteration, for the next item to be placed, the BL positions of this item with respect to all orientations are computed. Then one of the orientations is chosen based on some rules (e.g., the one with the lowest BL position is chosen), and the item is placed with the selected orientation at its BL position.

A natural way to generalize the best-fit algorithm is as follows: We are given a priority among all combinations of orientations and items. (Because there are four possible orientations, $0°$, $90°$, $180°$ and $270°$, there are $4n$ possible combinations.) Whenever an item is to be packed, the BL positions of all remaining items with respect to all orientations are computed, and then the combination that attains the left-most BL position among the lowest ones is chosen, resolving ties by choosing the one with the highest priority.

There might be other ways to generalize these algorithms, but the most expensive computation of such algorithms would be the computation of the BL positions of the remaining items with respect to all orientations. The data structures in Chapter 5 can be easily generalized to deal with this case just by preparing $4t$ copies of NFP layouts (and relevant data structures for each of them), each corresponding to a combination of an orientation and a shape in $\mathcal{T}$. Because this increases the computation time only

by a constant factor, the time complexity of the case with rotation is the same as the case without rotation. The case where the reflection of items is allowed can be treated similarly. Note that the same argument can also be applied to the case in which candidate orientations are different among items (e.g., item 1 cannot be rotated, item 2 can be rotated by 90°, 180° and 270°, item 3 can be rotated by 180° and so forth), and also to a more general case in which each item can take different shapes from a constant number of candidates.

Chapter 8

Conclusion

Throughout this thesis, we have considered the developments of construction heuristics for the rectilinear block packing problem. The results in this thesis are summarized as follows.

We first reviewed in Chapter 2 and 1 the background of packing problems and described the rectilinear block packing problem. There are numerous variants of packing problems and many of these problems are related to real-world applications. For the simplest version of packing problems, one-dimensional packing problems, many simple greedy algorithms perform well and exact algorithms can solve instances of considerable size in reasonable time. The packing problems with more than one dimension become much more complex and it is hard to solve them exactly even for small instances. However, in many industrial applications, e.g., the VLSI design, good solutions for large-scale instances are often necessary. To deal with such situations, we developed construction algorithms for the rectilinear block packing problem especially to find good solutions of large-scale instances in short time.

In Chapter 3, we explained several basic techniques and ideas used in our algorithms. First, we introduced some sophisticated data structures utilized in the implementations of our algorithms. We then explained the concept of the technique of no-fit polygon [7] that is very useful when determining whether two blocks overlap each other. We introduced two well-known construction heuristics for the rectangle packing problem and adopt the bottom-left strategy [9] as the main strategy of our construction algorithms.

Then, in Chapter 4, we generalized two well-known construction heuristics for the rectangle packing problem, the bottom-left and the best-fit algorithms, to solve the rectilinear block packing problem. We also gave an efficient implementations for these two algorithms. If naively implemented, the bottom-left algorithm requires $O(\min\{M^6, nm^3M^3\})$ time and the best-fit algorithm requires $O(nm^3M^3)$ time. We generalized the algorithm proposed

in [48] to find the BL position efficiently and reduced the running time of the bottom-left algorithm to $O(M^2 \log M)$ and that of the best-fit algorithm to $O(nmM \log M)$.

In Chapter 5, we proposed more efficient implementations of the bottom-left and the best-fit algorithms for the rectilinear block packing problem than those in Chapter 4. We designed sophisticated data structures that dynamically keep the information so that the BL position of each item can be found in sub-linear amortized time. We then analyzed the time complexities of the two construction algorithms and showed that both algorithms run in $O(mM \log M)$ time. We performed a series of experiments on a set of instances that are generated from nine benchmark instances to analyze the performance of our algorithms from both sides of occupation rate and running time. The computational results show that the proposed algorithms are especially efficient for large instances with repeated shapes. Even for instances with more than 10,000 rectilinear blocks with up to 60 distinct shapes, the proposed algorithms run in less than 12 seconds. The occupation rate of the packing layouts obtained by our algorithms often reached higher than 95% for large-scale instances. We compared the running time of our algorithms with the implementation explained in Chapter 4 and those with implementation proposed in Chapter 5. The computational results show that our efficient implementations significantly reduced the computation time of the bottom-left and the best-fit algorithms. We also created large-scale instances with up to 10,000 distinct shapes to test our efficient algorithms and showed that the running time increases almost linearly with the number of distinct shapes. For instances with 10,000 distinct shapes, our algorithms run in less than 6000 seconds.

In Chapter 6, we further analyzed the quality of packing layouts obtained by the bottom-left and the best-fit algorithm and proposed a new constructive heuristic algorithm, the partition-based best-fit (PBF) algorithm, for the rectilinear block packing problem. The PBF algorithm can be regarded as a bridge between the bottom-left and the best-fit algorithms and takes advantages of both of these two algorithms. We analyzed the time complexity of the proposed PBF algorithm and showed that it runs in $O(mM \log M)$ time with the efficient implementation explained in Chapter 5. We also proposed some effective rules utilized in the PBF algorithm and performed a series of experiments on 168 instances that were generated from nine benchmark instances. The occupation rate of the packing layouts obtained by the proposed PBF algorithm was more than 93% on average for these instances. The computational results show that the improvement on the performance of the occupation rate obtained by the PBF algorithm is remarkable compared with the bottom-left and the best-fit algorithms, and the PBF algorithm is especially effective for instances with many different sizes of shapes.

Finally, in Chapter 7, we briefly explained how to apply the efficient implementation to the rectilinear block packing problem with rotation. We show that the time complexity

of the case with rotation is the same as that without rotation.

In recent years, since the computers are getting faster, approaches based on simple local search and metaheuristics perform well for small-scale instances of NP-hard problems. However, development of efficient algorithms for the problems with complex structures or for the large-scale instances is still big challenge. In this thesis, we focused on the development of construction algorithms for the rectilinear block packing problem. We designed implementations for the algorithms utilizing sophisticated data structures to reduce the running time. As a consequence, our algorithms can obtain good solutions in short time for large-scale instances. The author hopes that the research in this thesis will be useful in practical applications and helpful for the development of efficient implementations of algorithms.

Bibliography

[1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization.* Princeton University Press, 2003.

[2] M. Adamowicz and A. Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8(1):27–33, 1976.

[3] M. Adamowicz and A. Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8(1):27–33, 1976.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

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

[6] R. Alvarez-Valdes, F. Parreno, and J. Tamarit. A branch and bound algorithm for the strip packing problem. *OR Spectrum*, 31(2):431–459, 2009.

[7] R. C. Art Jr. *An Approach to the Two Dimensional Irregular Cutting Stock Problem.* PhD thesis, Massachusetts Institute of Technology, 1966.

[8] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 1999.

[9] B. S. Baker, E. G. Coffman, Jr, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[10] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28(5):1130–1154, 1980.

[11] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[12] R. Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767–769, 1956.

[13] R. Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[14] J. Bennell, G. Scheithauer, Y. Stoyan, T. Romanova, and A. Pankratov. Optimal clustering of a pair of irregular objects. *Journal of Global Optimization*, 61(3):497–524, 2015.

[15] J. A. Bennell and K. A. Dowsland. Hybridising tabu search with optimisation techniques for irregular stock cutting. *Management Science*, 47(8):1160–1172, 2001.

[16] E. E. Bischoff and M. D. Marriott. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research*, 44(2):267–276, 1990.

[17] A. Bortfeldt and G. Wäscher. Constraints in container loading—a state-of-the-art review. *European Journal of Operational Research*, 229(1):1–20, 2013.

[18] E. Burke, R. Hellier, G. Kendall, and G. Whitwell. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research*, 54(3):587–601, 2006.

[19] E. K. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research*, 52(4):655–671, 2004.

[20] B. Chazelle. The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 100(8):697–707, 1983.

[21] D. Chen, J. Liu, Y. Fu, and M. Shang. An efficient heuristic algorithm for arbitrary shaped rectilinear block packing problem. *Computers & Operations Research*, 37(6):1068–1074, 2010.

[22] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[23] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[24] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2009.

[25] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957.

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

[27] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is FFD(I) $\leq \frac{11}{9}$OPT(I)+$\frac{6}{9}$. In B. Chen, M. Paterson, and G. Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614, pages 1–11. Springer, 2007.

[28] K. A. Dowsland and W. B. Dowsland. Solution approaches to irregular nesting problems. *European Journal of Operational Research*, 84(3):506–521, 1995.

[29] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[30] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

[31] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[32] K. Fujiyoshi, C. Kodama, and A. Ikeda. A fast algorithm for rectilinear block packing based on selected sequence-pair. *Integration, the VLSI Journal*, 40(3):274–284, 2007.

[33] K. Fujiyoshi and H. Murata. Arbitrary convex and concave rectilinear block packing using sequence-pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(2):224–233, 2000.

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

[35] F. Glover and G. A. Kochenberger, editors. *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2003.

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

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

[38] A. M. Gomes and J. F. Oliveira. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *European Journal of Operational Research*, 171(3):811–829, 2006.

[39] G. H. Gonner and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[40] T. F. Gonzalez, editor. *Handbook of Approximation Algorithms and Metaheuristics*. CRC Press, 2007.

[41] D. S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[42] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[43] E. Hopper and B. C. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research*, 128(1):34–57, 2001.

[44] E. Hopper and B. C. Turton. A review of the application of meta-heuristic algorithms to 2D strip packing problems. *Artificial Intelligence Review*, 16(4):257–300, 2001.

[45] T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization*. Baltzer, 1987.

[46] T. Ibaraki, K. Nonobe, and M. Yagiura, editors. *Metaheuristics: Progress as Real Problem Solvers*. Springer, 2005.

[47] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[48] S. Imahori, Y. Chien, Y. Tanaka, and M. Yagiura. Enumerating bottom-left stable positions for rectangle placements with overlap. *Journal of the Operations Research Society of Japan*, 57(1):45–61, 2014.

[49] S. Imahori and M. Yagiura. The best-fit heuristic for the rectangular strip packing problem: An efficient implementation and the worst-case approximation ratio. *Computers & Operations Research*, 37(2):325–333, 2010.

[50] S. Imahori, M. Yagiura, and T. Ibaraki. Local search algorithms for the rectangle packing problem with general spatial costs. *Mathematical Programming*, 97(3):543–569, 2003.

[51] S. Imahori, M. Yagiura, and T. Ibaraki. Improved local search algorithms for the rectangle packing problem with general spatial costs. *European Journal of Operational Research*, 167(1):48–67, 2005.

[52] P. Jain, P. Fenyes, and R. Richter. Optimal blank nesting using simulated annealing. *Journal of Mechanical Design*, 114(1):160–165, 1992.

[53] S. Jakobs. On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88(1):165–181, 1996.

[54] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[55] M. Kang and W. W. Dai. General floorplanning with L-shaped, T-shaped and soft blocks based on bounded slicing grid structure. In *Proceedings of the 1997 Asia and South Pacific Design Automation Conference*, pages 265–270. IEEE, 1997.

[56] M. Z. Kang and W.-M. Dai. Arbitrary rectilinear block packing based on sequence pair. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 259–266. IEEE, 1998.

[57] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3(1):59–71, 1999.

[58] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[59] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura, and H. Nagamochi. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198(1):73–83, 2009.

[60] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[61] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1998.

[62] C. Kodama and K. Fujiyoshi. Selected sequence-pair: An efficient decodable packing representation in linear time using sequence-pair. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 331–337. ACM, 2003.

[63] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, fifth edition, 2012.

[64] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[65] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher. Exhaustive approaches to 2D rectangular perfect packings. *Information Processing Letters*, 90(1):7–14, 2004.

[66] Z. Li and V. Milenkovic. Compaction and separation algorithms for non-convex polygons and their applications. *European Journal of Operational Research*, 84(3):539–561, 1995.

[67] J.-M. Lin, H.-L. Chen, and Y.-W. Chang. Arbitrarily shaped rectilinear module placement using the transitive closure graph representation. *IEEE Transactions on Very Large Scale Integration Systems*, 10(6):886–901, 2002.

[68] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–354. Kluwer Academic Publishers, 2003.

[69] Y. Ma, X. Hong, S. Dong, Y. Cai, S. Chen, C.-K. Cheng, and J. Gu. Arbitrary convex and concave rectilinear block packing based on corner block list. In *Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 5, pages V–493–V–496. IEEE, 2003.

[70] K. Machida and K. Fujiyoshi. The improvement of rectilinear block packing using sequence-par. Technical report, IEICE, VLD2000-134, (in Japanese), 2001.

[71] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, 15(3):310–319, 2003.

[72] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, 123(2):325–332, 2000.

[73] R. Martí. Multi-start methods. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 355–368. Kluwer Academic Publishers, 2003.

[74] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 472–479. IEEE, 1995.

[75] 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 of Integrated Circuits and Systems*, 15(12):1518–1524, 1996.

[76] H. Murata, K. Fujiyoshi, T. Watanabe, and Y. Kajitani. A mapping from sequence-pair to rectangular dissection. In *Proceedings of the 1997 Asia and South Pacific Design Automation Conference*, pages 625–633. IEEE, 1997.

[77] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module packing based on the BSG-structure and IC layout applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(6):519–530, 1998.

[78] S. Nakatake, M. Furuya, and Y. Kajitani. Module placement on BSG-structure with pre-placed modules and rectilinear modules. In *Proceedings of the 1998 Asia and South Pacific Design Automation Conference*, pages 571–576. IEEE, 1998.

[79] J. F. Oliveira, A. M. Gomes, and J. S. Ferreira. TOPOS–a new constructive algorithm for nesting problems. *OR Spectrum*, 22(2):263–284, 2000.

[80] I. H. Osman and J. P. Kelly, editors. *Meta-Heuristics: Theory & Applications*. Kluwer Academic Publishers, 1996.

[81] I. H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5):511–623, 1996.

[82] Y. Pang, C.-K. Cheng, K. Lampaert, and W. Xie. Rectilinear block packing using O-tree representation. In *Proceedings of the 2001 International Symposium on Physical Design*, pages 156–161. ACM, 2001.

[83] D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47(4):570–575, 1999.

[84] D. Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33(1):1–14, 1999.

[85] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[86] M. G. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–250. Kluwer Academic Publishers, 2003.

[87] K. Sakanushi, S. Nakatake, and Y. Kajitani. The multi-BSG: Stochastic approach to an optimum packing of convex-rectilinear blocks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 267–274. ACM, 1998.

[88] D. Simchi-Levi. New worst-case results for the bin-packing problem. *Naval Research Logistics*, 41:579–585, 1994.

[89] Y. Takashima and H. Murata. The tight upper bound of the empty rooms in floor-plan. *The 2001 Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 264–271, 2001.

[90] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[91] S. Voss, S. Martello, I. H. Osman, and C. Roucairol, editors. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, 1999.

[92] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[93] G.-M. Wu, Y.-C. Chang, and Y.-W. Chang. Rectilinear block placement using B*-trees. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):188–202, 2003.

[94] J. Xu, P.-N. Guo, and C.-K. Cheng. Rectilinear block placement using sequence-pair. In *Proceedings of the 1998 International Symposium on Physical Design*, pages 173–178. ACM, 1998.

A List of Author's Work

Journals

1. Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "Efficient Implementations of Construction Heuristics for the Rectilinear Block Packing Problem," *Computers and Operations Research*, 53 (2015) 206–222.

2. Y. Hu, H. Hashimoto, S. Imahori, T. Uno, M. Yagiura, "A Partition-Based Heuristic Algorithm for the Rectilinear Block Packing Problem," *Journal of the Operations Research Society of Japan*, 59 (2016) 110–129.

International Conferences with Review

1. Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "A New Construction Heuristic Algorithm for the Rectilinear Block Packing Problem: A Bridge between the Best-Fit and Bottom-Left Algorithms," *The IEEE International Conference on Industrial Engineering and Engineering Management* (2012) 182–186.

2. Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "Efficient Construction Heuristic Algorithms for the Rectilinear Block Packing Problem," *International Symposium on Scheduling* (2013) 80–85.

3. S. Fukatsu, Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "An Efficient Method for Checking Overlaps and Construction Algorithms for the Bitmap Shape Packing Problem," *The IEEE International Conference on Industrial Engineering and Engineering Management* (2014).

4. H. Iwasawa, Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "A Heuristic Algorithm for the Container Loading Problem of Challenge Renault/ESICUP," *International Symposium on Scheduling* (2015) 236–241.

5. K. Matsushita, Y. Hu, H. Hashimoto, S. Imahori, M. Yagiura, "An Exact Algorithm with Successively Strengthened Lower Bounds for the Rectilinear Block Packing Problem," *International Symposium on Scheduling* (2015) 242–247.

6. W. Wu, Y. Hu, H. Hashimoto, T. Ando, T. Shiraki, M. Yagiura, "A Heuristic Algorithm for the Crew Pairing Problem in Airline Scheduling," *International Symposium on Scheduling* (2015) 121–128.

7. Y. Takada, Y. Hu, H. Hashimoto, M. Yagiura, "An Iterated Local Search Algorithm for the Multi-Vehicle Covering Tour Problem," *The IEEE International Conference on Industrial Engineering and Engineering Management* (2015).