

Chapter 1

LOCAL SEARCH ALGORITHMS FOR THE TWO-DIMENSIONAL CUTTING STOCK PROBLEM WITH A GIVEN NUMBER OF DIFFERENT PATTERNS

Shinji Imahori,¹ Mutsunori Yagiura,¹ Shunji Umetani,² Shinya Adachi¹
and Toshihide Ibaraki¹

¹*Department of Applied Mathematics and Physics,
Graduate School of Informatics, Kyoto University*
{imahori,yagiura,shin,ibarak}@amp.i.kyoto-u.ac.jp

²*Department of Advanced Science and Technology,
Graduate School of Engineering, Toyota Technological Institute*
s-umetani@toyota-ti.ac.jp

Abstract We consider the two-dimensional cutting stock problem which arises in many applications in industries. In recent industrial applications, it is argued that the setup cost for changing patterns becomes more dominant and it is impractical to use many different cutting patterns. Therefore, we consider the pattern restricted two-dimensional cutting stock problem, in which the total number of applications of cutting patterns is minimized while the number of different cutting patterns is given as a parameter n . For this problem, we develop local search algorithms. As the size of the neighborhood plays a crucial role in determining the efficiency of local search, we propose to use linear programming techniques for the purpose of restricting the number of solutions in the neighborhood. In this process, to generate a cutting pattern, it is required to place all the given products (rectangles) in the stock sheet (two-dimensional area) without mutual overlap. For this purpose, we develop a heuristic algorithm using an existing rectangle packing algorithm with the sequence pair coding scheme. Finally, we generate random test instances of this problem and conduct computational experiments, to see the effectiveness of the proposed algorithms.

Keywords: Two-Dimensional Cutting Stock Problem, Linear Programming, Rectangle Packing, Neighborhood, Local Search

Introduction

We consider the two-dimensional cutting stock problem, which is one of the representative combinatorial optimization problems, and arises in many industries such as steel, paper, wood, glass and fiber. The problem can be defined as follows: We are given a sufficient number of stock sheets of the same width W and height H , and m types of rectangular products, where each product i has its width w_i , height h_i and demand d_i . From stock sheets we have to cut rectangular products whose number is specified as demands. The objective is to minimize the total number of stock sheets required. This problem is NP-hard, since this is a generalization of the two-dimensional bin packing problem and the (one-dimensional) cutting stock problem, which are already known to be NP-hard [5].

A classical approach to the one-dimensional cutting stock problem (1DCSP) is to formulate it as an integer programming problem (IP), and solve it by a heuristic method based on its linear programming (LP) relaxation. As it is impractical to consider all cutting patterns which correspond to columns in the LP relaxation, Gilmore and Gomory [6,7] proposed a column generation technique that generates only the columns necessary to improve the lower bound of IP by solving the associated knapsack problems. The LP relaxation often has a property that the round up of the LP lower bound is equal to the optimal value of IP [14]. Based on these ideas, branch-and-bound, heuristic and metaheuristic algorithms with the column generation technique have been developed with certain computational success [20].

It is however observed that those approaches tend to use many (very close to the number of product types) different cutting patterns. In recent cutting industries, the setup cost for changing patterns is more significant and it is often impractical to use many different cutting patterns. Several researchers (e.g., Foerster and Wäscher [4], Haessler [9], Umetani et al. [17]) have proposed algorithms for 1DCSP with consideration on the number of cutting patterns.

The two-dimensional cutting stock problem (2DCSP) has also been extensively studied. Gilmore and Gomory [8] extended their previous work on 1DCSP [6,7] to the two-dimensional case. They proposed a column generation scheme in which new columns (new cutting patterns) are produced by solving the generalized knapsack problems. After this, various heuristic algorithms based on the column generation technique

have been proposed [1,3,19,21]. Nevertheless, to the authors' knowledge, there is no study on 2DCSP under the constraints on the number of different patterns.

In this chapter, we consider the two-dimensional cutting stock problem using a given number of different patterns n (we call this problem 2DCSP n). 2DCSP n asks to determine a set of cutting patterns, whose size is n or less, and the numbers of applications of each cutting pattern. The objective is to minimize the total number of applications of cutting patterns.

The problem of deciding the number of applications for each pattern becomes an integer programming problem (IP). In Section 2, we propose a heuristic algorithm for the IP, which is based on its linear programming (LP) relaxation. We incorporate a sensitive analysis technique and the criss-cross method [22], a variant of simplex method, into our algorithm for efficiency. In Section 3, we propose local search algorithms to find a good set of cutting patterns. As the size of the neighborhood plays a crucial role in determining the efficiency of local search, we propose to utilize the dual solution of the LP relaxation for the purpose of restricting the number of solutions in the neighborhood. We design two kinds of neighborhood, basic and enhanced. Then they are computationally compared from the view point of the performance of the resulting local search algorithms.

To generate a feasible cutting pattern, we have to place all the given products in the stock sheet (two-dimensional area) without mutual overlap. At this placement stage, we assume that each product can be rotated by 90° , and assume no constraint on products' placement such as "guillotine cut". In Section 4, we first propose simple methods to check the feasibility of a given set of products. That is, under some conditions, we could easily find a feasible placement for a given pattern quickly or its infeasibility. In general, however, the problem to place all the products in a stock sheet without mutual overlap is NP-hard. In order to find a feasible placement, we use a local search algorithm developed for the rectangle packing problem [10,11] with the sequence pair coding scheme [15]. It is however computationally too expensive if we always use the original rectangle packing algorithm, since we must solve the problem many times. Therefore, we modify it to a faster heuristic algorithm.

In Section 6, we generate random test instances of 2DCSP and conduct computational experiments to compare our algorithms with various different neighborhood operations. We also compute the trade-off curve between the number of different cutting patterns n and the solution quality.

1. Problem

To define the two-dimensional cutting stock problem (2DCSP), we are given a sufficient number of stock sheets of the same width W and height H , and m types of rectangular products $M = \{1, 2, \dots, m\}$, where each product i has its width w_i , height h_i and demand d_i . A cutting pattern p_j is described as $p_j = (a_{1j}, a_{2j}, \dots, a_{mj})$, where $a_{ij} \in \mathbf{Z}_+$ (the set of nonnegative integers) is the number of product i cut from a stock sheet by pattern p_j . A placement of products in a pattern is a set of their locations in one stock sheet together with their orientations (i.e., the original direction or rotated by 90°), where a placement is feasible if all the products are placed in one stock sheet without mutual overlap. We call a pattern p_j feasible if it has a feasible placement. Let S denote the set of all feasible patterns. Note that, the set S is very large and it is not explicitly given; i.e., we must find a feasible placement to confirm that a pattern is feasible.

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

$$\begin{aligned}
 \text{2DCSP:} \quad & \text{minimize} && f(\Pi, X) = \sum_{p_j \in \Pi} x_j && (1.1) \\
 & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in M, \\
 & && \Pi \subseteq S, \\
 & && x_j \in \mathbf{Z}_+, \text{ for } p_j \in \Pi.
 \end{aligned}$$

Here we consider a variant of 2DCSP with an input parameter n , where n is the number of different cutting patterns $|\Pi|$. We call this problem the two-dimensional cutting stock problem with a given number

of different patterns n (2DCSPn), which is formally defined as follows:

$$\begin{aligned}
\text{2DCSPn:} \quad & \text{minimize} && f(\Pi, X) = \sum_{p_j \in \Pi} x_j && (1.2) \\
& \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in M, \\
& && \Pi \subseteq S, \\
& && |\Pi| \leq n, \\
& && x_j \in \mathbf{Z}_+, \text{ for } p_j \in \Pi.
\end{aligned}$$

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

$$L_1 = \left\lceil \sum_{i \in M} d_i w_i h_i / WH \right\rceil. \quad (1.3)$$

This bound is easy to compute (it can be computed in $O(m)$ time), and is a good bound when there are many products of small sizes. We also introduce another lower bound L_2 , which works effective if there are many large products. This lower bound is obtained by concentrating on large products, and is a little complicated since each product can be rotated by 90° . We note that various lower bounds are known for the two-dimensional bin packing problem without rotation [14].

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

$$\begin{cases}
w_i^* = h_i^* = \min\{w_i, h_i\} & (\text{if } \max\{w_i, h_i\} \leq \min\{W, H\}), \\
w_i^* = w_i, h_i^* = h_i & (\text{if } w_i > H \text{ or } h_i > W), \\
w_i^* = h_i, h_i^* = w_i & (\text{if } w_i > W \text{ or } h_i > H).
\end{cases} \quad (1.4)$$

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

$$\begin{aligned}
M^W &= \{i \in M : w_i^* > W/2\}, \\
M_1^W(q) &= \{i \in M^W : h_i^* > H - q\}, \\
M_2^W(q) &= \{i \in M^W : H - q \geq h_i^* \geq q\}.
\end{aligned} \quad (1.5)$$

No two products $i \in M_1^W(q)$ and $j \in \{M_1^W(q) \cup M_2^W(q)\}$ can be packed into the same stock sheet, and at most $\lfloor H/q \rfloor$ products in $M_2^W(q)$ can be packed into one stock sheet. Therefore, we have the following lower bound:

$$L_2^W = \max_{0 < q \leq H/2} \{ |M_1^W(q)| + \lceil |M_2^W(q)| / \lfloor H/q \rfloor \rceil \}. \quad (1.6)$$

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

Algorithm Compute L_2^W

Step 1: Divide all products into two sets M^W and $I \setminus M^W$. For each product $i \in M^W$, if $h_i^* \leq H/2$, set $s(i) := h_i^*$; otherwise set $s(i) := H - h_i^* + \varepsilon$. Sort rectangles $i \in M^W$ in the ascending order of $s(i)$. Set $L_2^W := 0, q := 0, |M_1^W(q)| := 0, M_2^W(q) := M^W$ and $|M_2^W(q)| := |M^W|$.

Step 2: Choose a rectangle $i \in M_2^W(q)$ with the smallest $s(i)$. If $h_i^* \leq H/2$, go to Step 3; Otherwise go to Step 4.

Step 3: Set $q := h_i^*$ and compute $|M_1^W(q)| + \lceil |M_2^W(q)| / \lfloor H/q \rfloor \rceil$. If this is larger than L_2^W , set $L_2^W := |M_1^W(q)| + \lceil |M_2^W(q)| / \lfloor H/q \rfloor \rceil$. Set $M_2^W(q) := M_2^W(q) \setminus \{i\}$ and $|M_2^W(q)| := |M_2^W(q)| - 1$. If $M_2^W(q) = \emptyset$, output L_2^W and halt; otherwise return to Step 2.

Step 4: Set $q := H - h_i^* + \varepsilon$, $M_2^W(q) := M_2^W(q) \setminus \{i\}, |M_2^W(q)| := |M_2^W(q)| - 1$ and $|M_1^W(q)| := |M_1^W(q)| + 1$, and compute $|M_1^W(q)| + \lceil |M_2^W(q)| / \lfloor H/q \rfloor \rceil$. If this is larger than L_2^W , set $L_2^W := |M_1^W(q)| + \lceil |M_2^W(q)| / \lfloor H/q \rfloor \rceil$. If $M_2^W(q) = \emptyset$, output L_2^W and halt; otherwise return to Step 2.

We also define $M^H, M_1^H(q)$ and $M_2^H(q)$ in the same manner, and another lower bound L_2^H is similarly computed. As a result, the following lower bound L_2 is derived by considering large products only.

$$L_2 = \max \{L_2^W, L_2^H\}. \quad (1.7)$$

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

$$f_{LB} = \max \{L_1, L_2\}. \quad (1.8)$$

2. Computing the number of pattern applications

In this section, we consider the problem of computing $X = (x_1, x_2, \dots, x_n)$ for a given set of patterns $\Pi = \{p_1, p_2, \dots, p_n\}$, where x_j denotes the number of applications of pattern p_j . This problem is described as the

following integer programming problem:

$$\begin{aligned}
 \text{IP}(\Pi) : \quad & \text{minimize} && f(X) = \sum_{p_j \in \Pi} x_j && (1.9) \\
 & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in M, \\
 & && x_j \in \mathbf{Z}_+, \text{ for } p_j \in \Pi.
 \end{aligned}$$

This problem is already known to be NP-hard since x_j must be integer. There are several studies for 2DCSP that solve this problem exactly with branch-and-bound method. However, as we must solve this problem many times in our algorithm, we consider a faster heuristic algorithm.

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

$$\begin{aligned}
 \text{LP}(\Pi) : \quad & \text{minimize} && f(X) = \sum_{p_j \in \Pi} x_j && (1.10) \\
 & \text{subject to} && \sum_{p_j \in \Pi} a_{ij} x_j \geq d_i, \text{ for } i \in M, \\
 & && x_j \geq 0, \text{ for } p_j \in \Pi.
 \end{aligned}$$

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

In our heuristic algorithm, we do a little more than the simple rounding up. We first set $\hat{x}_j := \lfloor \bar{x}_j \rfloor$ for all patterns $p_j \in \Pi$ and let $\hat{X} = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_n)$. We then sort all indexes j in the descending order of $\bar{x}_j - \lfloor \bar{x}_j \rfloor$, and round up \bar{x}_j to $\lceil \bar{x}_j \rceil$ or round down \bar{x}_j to $\lfloor \bar{x}_j \rfloor$ in the resulting order of j according to the following rule. If pattern p_j includes a product i such that $\sum_j a_{ij} \hat{x}_j < d_i$, we set $\hat{x}_j := \hat{x}_j + 1$ and go to the next pattern. If pattern p_j does not include such products, we do not change \hat{x}_j (i.e., rounding down) and go to the next. The solution \hat{X} obtained by this process is always feasible to IP(II) and not worse than the solution obtained by the simple rounding up method. Note that various rounding procedures are compared in e.g., Valdés et al. [18], Vanderbeck [20].

In our local search algorithm for finding a good set of patterns, which will be explained in the next section, we must solve many LP(II). If LP is naively solved from scratch whenever we evaluate a new set of patterns, the computation becomes expensive. Therefore, as in our previous

paper [17], we incorporate a sensitive analysis technique and the criss-cross method [22], a variant of simplex method, to utilize an optimal LP solution for the current set of patterns Π . For each set of patterns $\Pi' \in N(\Pi)$, where $N(\Pi)$ is the family of neighborhood sets of patterns of Π , the criss-cross algorithm converges to an optimal solution of $LP(\Pi')$ usually after a few pivot operations.

3. A local search algorithm to find a good set of patterns

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

The following ingredients must be specified in designing LS: Initial solution, neighborhood, move strategy and a function to evaluate solutions. First, we will explain how to prepare an initial feasible solution, and then we will design several neighborhoods. As the move strategy, we adopt the first admissible move strategy (i.e., as soon as we find a better solution in its neighborhood, we move to the solution). A set of patterns Π is evaluated by the optimal value of LP relaxation $LP(\Pi)$. Note that, we also compute an integer solution \hat{X} of $IP(\Pi)$ heuristically by the algorithm in Section 2, and update the incumbent solution accordingly (i.e., the best feasible solution among those obtained so far).

Initial solution

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

In order to construct a feasible solution, we ignore the demand d_i of product i , and temporarily we assume it as one. Therefore, we consider

the two-dimensional bin packing problem (2DBPP) instead of 2DCSP n .

$$\begin{aligned}
 \text{2DBPP:} \quad & \text{Find a set of patterns } \Pi, & (1.11) \\
 & \text{subject to } \sum_{p_j \in \Pi} a_{ij} \geq 1, \text{ for } i \in M, \\
 & \Pi \subseteq S, \\
 & |\Pi| \leq n.
 \end{aligned}$$

If a feasible solution Π for 2DBPP is given, it is easy to construct a feasible solution for 2DCSP n by using appropriately large numbers of applications x_j of cutting patterns. We utilize a heuristic algorithm for 2DBPP which is based on the next-fit algorithm known for the one-dimensional bin packing problem. Note that, 2DBPP has been well studied and there are more sophisticated heuristic and metaheuristic algorithms [12].

Neighborhoods

A neighborhood $N(\Pi)$ may be naturally defined by replacing each cutting pattern p_j in the set Π with another cutting pattern $p'_j \in S \setminus \Pi$:

$$N(\Pi) = \{ \Pi \cup \{p'_j\} \setminus \{p_j\} \mid p_j \in \Pi, p'_j \in S \setminus \Pi \}, \quad (1.12)$$

where S is the set of all feasible cutting patterns. As mentioned in Section 1, the number of all feasible cutting patterns $|S|$ is too large, and most of them may not lead to improvement. In view of this, we propose heuristic algorithms to generate smaller neighborhoods.

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

$$r_i = \sum_j a_{ij} x_j - d_i \quad (1.13)$$

of product i ; we sort all products in the descending order of r_i , and then remove the products in this order. This is because, it is impossible to

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

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

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

Replacement operation. If all products in pattern p_j are removed by the redundancy reduction and basic operations, we must reconstruct a new pattern from scratch by the basic and filling-up operations. This situation always occurs for pattern p_j with $x_j = 0$, and such reconstruction may not find a pattern with small trim loss. Therefore, we replace

p_j with a new pattern in this case. For this purpose, we keep cn (c is a parameter and we use $c = 3$) good cutting patterns obtained by then in memory, and choose one from them, where we define good cutting pattern as those having small trim loss. We call this as the replacement operation.

Enhanced neighborhood. Now, our new neighborhood is the set of solutions obtained from Π by applying the operations proposed in this section (i.e., basic, redundancy reduction, filling-up and replacement). We call this neighborhood as the enhanced neighborhood. Two neighborhoods, basic and enhanced, will be computationally compared in Section 6.

4. Feasibility check for a pattern

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

$$\sum a_{ij}l_i \leq L, \quad (1.14)$$

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

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

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

$$\sum_{i \in M} a_{ij}w_ih_i \leq WH. \quad (1.15)$$

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

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

Rectangle Packing Problem

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

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

There are many heuristic algorithms in the literature proposed for this problem. Basic ingredients of such algorithms are as follows.

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

In this chapter, we use a coding scheme called sequence pair [15]. A sequence pair is a pair of permutations $\sigma = (\sigma_+, \sigma_-)$ of given rectangles where $\sigma_+(\alpha) = k$ (equivalently $\sigma_+^{-1}(k) = \alpha$) means that rectangle k is the α th rectangle in σ_+ (σ_- is similarly defined). Given a sequence pair $\sigma = (\sigma_+, \sigma_-)$, two partial orders \preceq_σ^x and \preceq_σ^y are defined by

$$\begin{aligned} \sigma_+^{-1}(k) < \sigma_+^{-1}(k') \text{ and } \sigma_-^{-1}(k) < \sigma_-^{-1}(k') &\iff k \preceq_\sigma^x k', \\ \sigma_+^{-1}(k) > \sigma_+^{-1}(k') \text{ and } \sigma_-^{-1}(k) < \sigma_-^{-1}(k') &\iff k \preceq_\sigma^y k', \end{aligned} \quad (1.16)$$

for any pair k and k' ($k \neq k'$) of rectangles. Based on these partial orders, we impose the following constraints on relative locations:

$$\begin{aligned} k \preceq_\sigma^x k' &\implies l_k^x + w_k \leq l_{k'}^x, \\ k \preceq_\sigma^y k' &\implies l_k^y + h_k \leq l_{k'}^y, \end{aligned} \quad (1.17)$$

where w_k and h_k are the current width and height of rectangle k (depending on its orientation), and the location (l_k^x, l_k^y) is the x and y coordinates of its lower left corner. That is, if $k \preceq_\sigma^x k'$ holds, the right side of rectangle k should be placed to the left of the left side of rectangle k' . If $k \preceq_\sigma^y k'$ holds, the upper hem of rectangle k should be placed to the below of the lower hem of rectangle k' . This coding scheme has following properties: (1) Constraints (1.17) are a sufficient condition for having no mutual overlap. (2) It is possible to find a feasible placement in polynomial time for a given sequence pair σ , if there exists a feasible placement which satisfies constraints (1.17).

In [15], Murata et al. proposed a decoding algorithm (i.e., an algorithm to compute locations of all rectangles under constraints (1.17) from a given sequence pair σ) which runs in $O(s^2)$ time, where s is the number of rectangles to place. After this, the time complexity has been improved; in [10], Imahori et al. proposed a decoding algorithm with $O(s \log s)$ running time, and later it was improved to $O(1)$ amortized computational time per one sequence pair if used in the neighborhood search [11]. Note that, the last algorithm outputs only feasibility; they need $O(s \log s)$ more time if a feasible placement for σ is required.

There are several local search or metaheuristic algorithms proposed to find a good sequence pair [10,11,15]. A local search algorithm proposed by Imahori et al. [11] is one of the most efficient ones among them. We call this the original algorithm in this chapter. As it is too expensive to use the original algorithm in our local search, we modify it in the following manner.

To construct an initial solution for the local search, we apply the original rectangle packing algorithm for each pattern in Π . In our neighborhood search, we modify a pattern $p_j \in \Pi$ to another pattern p'_j . Since we already have a good sequence pair σ for pattern p_j , we construct a sequence pair σ' for p'_j starting from σ . As noted before, we remove some products from pattern p_j and add some other products to construct a new pattern p'_j . When a product i is removed, we just remove it from the current sequence pair σ . If p_j includes two or more units of product i (i.e., $a_{ij} \geq 2$), we remove one of them at random. When we add a new product i' , we check all insertions of i' to all positions of the sequence pair. The number of sequence pairs we check in this process is s^2 , where s is the number of products in pattern p_j (i.e., $s = \sum_i a_{ij}$). We check all of them in $O(s^2)$ time with the algorithm Evaluate-Shift-Moves proposed in [11]. When there is more than one product to be added, we add them one by one to the sequence pairs generated by then.

5. The entire algorithm

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

Algorithm $\text{NS}((\Pi, \overline{X}), (\Pi^*, \hat{X}^*), p_j, t)$, where NS stands for neighborhood search, is the core routine of our entire algorithm, which is comprised of those algorithms described in Sections 2,3 and 4. Starting from the current set of patterns Π , this algorithm checks the family of sets of patterns generated by modifying the pattern $p_j \in \Pi$ by the basic, redundancy reduction, filling-up and replacement operations, for a given parameter $t (= 0, 1 \text{ or } 2)$ used in the basic operation. It also computes

an integer solution for each set of patterns by the heuristic algorithm of Section 2, and updates the incumbent solution (Π^*, \hat{X}^*) if a better integer solution is obtained.

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

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

Output: Check if an improved solution (Π', \bar{X}') exists, while updating the incumbent solution (Π^*, \hat{X}^*) .

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

Step 2: Repeat the following procedure cn times (where cn is the number of good cutting patterns in memory, as described in Section 3), and then go to Step 5.

(Replacement operation): Replace pattern p_j with a good cutting pattern stored in memory. Compute IP solution (Π', \hat{X}') and LP solution (Π', \bar{X}') for the resulting set of patterns as described in Section 2. If (Π', \hat{X}') is better than the incumbent solution (Π^*, \hat{X}^*) , update it. If (Π', \bar{X}') is better than the current solution (Π, \bar{X}) , exit with (Π', \bar{X}') .

Step 3 (redundancy reduction):

Remove the redundant products from p_j as many as possible, by the procedure in Section 3. Denote the resulting pattern as p_j^1 . If this pattern does not have more than t products, go to Step 2; otherwise go to Step 4.

Step 4: Apply the following procedures to every subset of products M' with $|M'| = t$ and every product $i \notin M'$ with $r_i = 0$, as described in Section 3.

4-1 (basic operation):

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

4-2 (filling-up operation):

Fill-up the pattern p_j^2 by the procedure in Section 3. Compute IP solution (Π', \hat{X}') and LP solution (Π', \bar{X}') for the resulting

set of patterns as described in Section 2. If (Π', \hat{X}') is better than the incumbent solution (Π^*, \hat{X}^*) , update it. If (Π', \overline{X}') is better than the current solution (Π, \overline{X}) , exit with (Π', \overline{X}') .

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

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

Algorithm LS_2DCSP n

```

Line 1: Construct an initial set of patterns  $\Pi$  and compute its LP solution  $\overline{X}$ ;
Line 2: Set  $\Pi^* := \Pi$  and compute its IP solution  $\hat{X}^*$  (i.e., the incumbent solution);
Line 3: Start the neighborhood search from the current solution  $(\Pi, \overline{X})$ ;
Line 4:   for  $t = 0, 1, 2$  do
Line 5:     for  $p_j \in \Pi$  do
Line 6:       NS( $(\Pi, \overline{X}), (\Pi^*, \hat{X}^*), t, p_j$ ) to obtain an improved solution;
Line 7:       if an improved solution  $(\Pi', \overline{X}')$  is found then
Line 8:         set  $(\Pi, \overline{X}) := (\Pi', \overline{X}')$  and return to Line 3;
Line 9:       end for
Line 10:    end for
Line 11: Output the incumbent solution  $(\Pi^*, \hat{X}^*)$  and halt;

```

6. Computational experiments

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

Other existing algorithms

In the literature, several heuristic algorithms have been proposed for 2DCSP [1,2,3,18,19,21] and some of their computational results have been reported. For the evaluation of our algorithm, it is desirable to compare our algorithm with such computational results. However, it is not easy for the following reasons.

First of all, there are many variations of 2DCSP and those algorithms in the literature were designed for slightly different problems. Cung et al. [2] and Valdés et al. [18] considered the following problem, and proposed branch-and-bound and heuristic algorithms: Cut a single rectangular stock sheet into a set of small rectangular products of given sizes and values so as to maximize the total value. If the value of each product is equal to its area, the objective is to minimize the trim loss. Valdés et al. considered another problem in [19]: A set of stock sheets

of different sizes and a set of rectangular products are given. Each product has its width, height, demand and a fixed orientation. From these stock sheets, products are cut by “guillotine cut” in order to satisfy all demands. The objective is to minimize the total area of stock sheets required. Vanderbeck [21] proposed a heuristic algorithm, based on a nested decomposition for 2DCSP with various constraints such as 3-stage pattern and the maximum number of products in one pattern. Chauny and Loulou [1] and Farley [3] considered a similar problem to ours, except that the number of different cutting patterns n is specified in our problem. In [1] and [3], heuristic algorithms based on the column generation technique were proposed, together with some computational results. However, their computational results are too limited to compare.

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

Test instances

We generated random test instances of 2DCSP following the generation scheme described in [16,19]. The instances are characterized by the following three parameters.

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

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

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

Hence, there are 60 types of instances and we generated one instance for each type. These instances are named like “20S α ”, “20S β ”, ..., “20S ε ”, “20L α ”, ..., “20V ε ”, “30S α ”, ..., “50V ε ”. In our computational experiments, we apply our local search algorithms ten times to each instance with different initial solutions, and report the average results of ten trials. All test instances are electronically available from our web site (<http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing/>).

Comparison of basic and enhanced neighborhoods

First, basic and enhanced neighborhoods were computationally compared. For each instance, we applied our local search algorithm with each type of neighborhood ten times, and report the average quality of

the obtained solutions and computational time, where local search halts only when a locally optimal solution is reached. For simplicity, we set the number of different cutting patterns to the number of product types (i.e., $n = m$). Results are shown in Table 1.1. Column “m” shows the

Table 1.1. Comparison two neighborhoods in solution quality and computational time

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

number of product types. For each m , we have 15 instances with different ranges of demands and different sizes of stock sheet; e.g., we have instances $20S\alpha, 20S\beta, \dots, 20V\epsilon$ for $m = 20$. Column “quality” shows the average of the following ratio,

$$\text{quality} = 100 \cdot (f - f_{LB}) / f_{LB}, \quad (1.18)$$

where f is the number of stock sheets used in the solution, and f_{LB} is a lower bound of the number of required stock sheets computed by (1.8) in Section 1. The smaller quality means the better performance of the algorithm. Column “time” shows the average CPU time in seconds of one local search. These notations are also used in Table 1.2.

From Table 1.1, we can observe that the enhanced neighborhood gives smaller quality value than the basic neighborhood in all cases, while using similar computational time. It indicates that the redundancy reduction, filling-up and replacement operations proposed in Section 3 make the search more powerful and efficient. Based on this, we will use the enhanced neighborhood in the following experiments.

Effect of the number of patterns n

Next, we conducted computational experiments for different number of patterns n , i.e., n was set to $m, 0.8m, 0.6m$ and $0.4m$. Results are given in Table 1.2. The leftmost column shows the instance classes. For example, “class 20” represents 15 instances with $m = 20$. Each figure in this row is the average of 150 trials (that is, 10 trials with different initial solutions for each instance, and there are 15 instances for class 20). “class S” represents 20 instances whose demand is taken from interval

Table 1.2. Quality and time with various number of different patterns on various classes

	$n = m$		$n = 0.8m$		$n = 0.6m$		$n = 0.4m$	
	quality	time	quality	time	quality	time	quality	time
class 20	10.491	18.424	12.247	6.308	14.432	3.753	20.810	1.954
class 30	8.707	45.758	9.899	18.533	12.175	6.424	16.758	3.098
class 40	8.758	144.932	10.360	45.330	12.218	15.630	16.891	7.091
class 50	8.177	636.861	9.940	143.135	11.504	37.292	15.315	12.117
class S	12.531	149.200	14.555	47.335	16.377	15.373	20.769	6.348
class L	6.185	262.195	7.390	57.178	9.100	16.865	12.403	5.369
class V	8.384	223.087	9.890	55.467	12.269	15.086	19.158	6.478
class α	10.516	46.944	12.436	10.001	16.323	1.843	28.203	0.264
class β	8.048	71.712	10.027	15.579	12.269	3.388	19.436	0.780
class γ	7.924	141.175	9.661	28.295	11.530	6.258	14.912	1.627
class δ	7.331	311.113	8.483	60.768	9.397	15.827	10.795	6.567
class ε	11.348	486.525	12.451	151.9901	13.391	51.559	13.871	21.087
average	9.033	211.494	10.612	53.327	12.582	15.775	17.443	6.065

[1, 25], and each figure is the average of 200 trials. Other rows can be similarly interpreted. Now from the rows for classes 20, 30, 40 and 50 in Table 1.2, we can observe that as m becomes larger (i.e., from class 20 to 50), computational time increases and solution quality becomes slightly better. As n becomes smaller, the size of neighborhood becomes smaller and local search algorithm converges to locally optimal solutions rather quickly, making the quality of obtained solution poorer.

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

From the rows for different sizes of stock sheet (i.e., class $\alpha, \beta, \gamma, \delta$ and ε), we can observe that the solution qualities for classes α and ε are worse than others. The reason for class ε is similar to the previous one. For many test instances of class ε , we could find good solutions if the numbers of applications x_j can be fractional. However, these solutions degrade after obtaining integer solutions. On the other hand, as the size of stock sheet becomes smaller, it becomes harder to find a placement

of products with small unused area, since there are not enough small products to fill up the stock sheet.

Trade-off curve between n and solution quality

Finally, we conducted more detailed experiment to obtain the trade-off curve between n and the quality of the obtained solutions. We used two instances $40V\alpha$ and $40V\delta$. The area of the stock sheet of $40V\delta$ is four times as large as that of $40V\alpha$. Results are shown in Figures 1.1 and 1.2. In these figures, horizontal axis is n , and vertical axis shows the solution quality and CPU time in seconds. $40V\alpha$ -LP (resp., $40V\delta$ -LP) shows the average quality of obtained LP solution (i.e., the numbers of applications can be fractional) for $40V\alpha$ (resp., $40V\delta$), and $40V\alpha$ -IP (resp., $40V\delta$ -IP) shows the average quality of obtained IP solution (i.e., the numbers of applications must be integer) for $40V\alpha$ (resp., $40V\delta$). $40V\alpha$ -time and $40V\delta$ -time show the average CPU times in seconds for ten trials. When n is very small (i.e., $n \leq 11$ for $40V\alpha$ and $n \leq 2$ for $40V\delta$), we could not find initial feasible solutions. From Figures 1.1 and 1.2, we observe that the computational time tends to increase and the solution quality improves as n increases. For larger n , the improvement in quality becomes tiny instead of the computational time is increasing steadily. Note that, if the numbers of applications can be fractional, it is known that an optimal solution for 2DCSP uses at most m different patterns. Nevertheless, our obtained LP solutions for these instances with $n = 40$ are slightly worse than those with larger n . From these observations, there is still room for improvement in our neighborhood search. We also observe that the gap between LP and IP solutions for $40V\delta$ are more significant than the gap for $40V\alpha$.

7. Conclusion

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

To see the performance of our local search algorithm, we conducted some computational experiments with randomly generated test instances of 2DCSP. We first confirmed the effectiveness of our enhanced neighborhood, which utilized basic, redundancy reduction, filling-up and re-

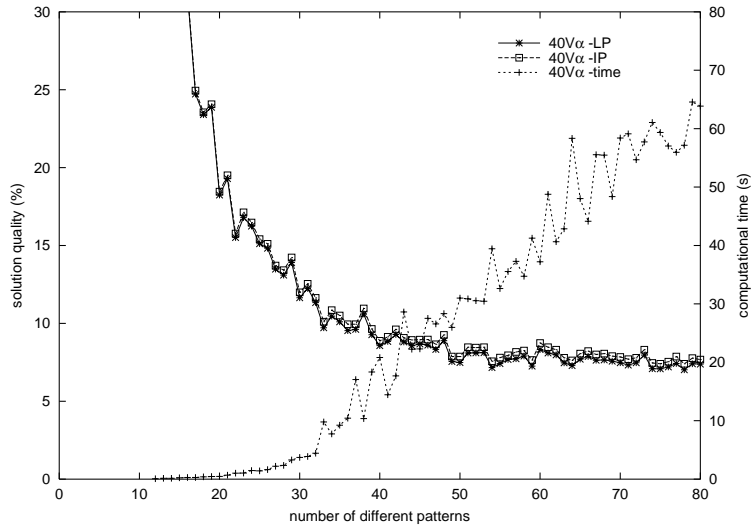


Figure 1.1. Trade-off between n and solution quality for $40V\alpha$

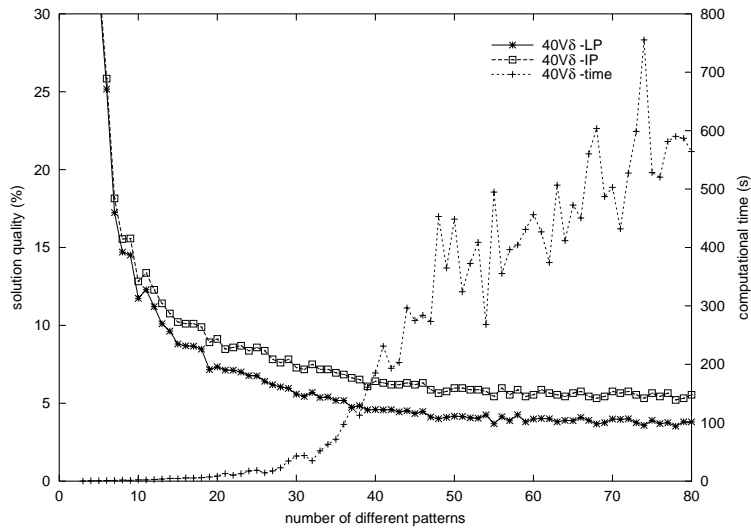


Figure 1.2. Trade-off between n and solution quality for $40V\delta$

placement operations. We also computed the trade-off curves between n and the quality of the obtained solution.

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

References

- [1] F. Chauny and R. Loulou, “LP-based method for the multi-sheet cutting stock problem,” *INFOR*, Vol. 32, 1994, pp. 253–264.
- [2] V.D. Cung, M. Hifi and B.L. Cun, “Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm,” *International Transactions in Operational Research*, Vol. 7, 2000, pp. 185–210.
- [3] A.A. Farley, “Practical adaptations of the Gilmore-Gomory approach to cutting stock problems,” *OR Spectrum*, Vol. 10, 1998, pp. 113–123.
- [4] H. Foerster and G. Wäscher, “Pattern reduction in one-dimensional cutting stock problems,” *International Journal of Production Research*, Vol. 38, 2000, pp. 1657–1676.
- [5] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, 1979).
- [6] P.C. Gilmore and R.E. Gomory, “A linear programming approach to the cutting-stock problem,” *Operations Research*, Vol. 9, 1961, pp. 849–859.
- [7] P.C. Gilmore and R.E. Gomory, “A linear programming approach to the cutting-stock problem-Part II,” *Operations Research*, Vol. 11, 1963, pp. 863–888.
- [8] P.C. Gilmore and R.E. Gomory, “Multistage cutting stock problems of two and more dimensions,” *Operations Research*, Vol. 13, 1965, pp. 94–120.
- [9] R.E. Haessler, “Controlling cutting pattern changes in one-dimensional trim problems,” *Operations Research*, Vol. 23, 1975, pp. 483–493.
- [10] S. Imahori, M. Yagiura and T. Ibaraki, “Local search algorithms for the rectangle packing problem with general spatial costs,” *Mathematical Programming*, Vol. 97, 2003, pp. 543–569.
- [11] S. Imahori, M. Yagiura and T. Ibaraki, “Improved local search algorithms for the rectangle packing problem with general spatial costs,” submitted for publication (available at <http://www-or.amp.i.kyoto-u.ac.jp/~imahori/packing>).
- [12] A. Lodi, S. Martello and D. Vigo, “Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems,” *Inform Journal on Computing*, Vol. 11, 1999, pp. 347–357.

- [13] O. Marcotte, “The cutting stock problem and integer rounding,” *Mathematical Programming*, Vol. 33, 1985, pp. 82–92.
- [14] S. Martello and D. Vigo, “Exact solution of the two-dimensional finite bin packing problem,” *Management Science*, Vol. 44, 1998, pp. 388–399.
- [15] 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*, Vol. 15, 1996, pp. 1518–1524.
- [16] J. Riehme, G. Scheithauer and J. Terno, “The solution of two-stage guillotine cutting stock problems having extremely varying order demands,” *European Journal of Operational Research*, Vol. 91, 1996, pp. 543–552.
- [17] S. Umetani, M. Yagiura and T. Ibaraki, “An LP-based local search to the one dimensional cutting stock problem using a given number of cutting patterns,” *IEICE Transactions on Fundamentals*, Vol. E86-A, 2003, pp. 1093–1102.
- [18] R.A. Valdés, A. Parajón, and J.M. Tamarit, “A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems,” *Computers & Operations Research*, Vol. 29, 2002, pp. 925–947.
- [19] R.A. Valdés, A. Parajón, and J.M. Tamarit, “A computational study of LP-based heuristic algorithms for two-dimensional guillotine cutting stock problems,” *OR Spectrum*, Vol. 24, 2002, pp. 179–192.
- [20] F. Vanderbeck, “Computational study of a column generation algorithm for bin packing and cutting stock problems,” *Mathematical Programming*, Vol. 86, 1999, pp. 565–594.
- [21] F. Vanderbeck, “A nested decomposition approach to a three-stage, two-dimensional cutting-stock problem,” *Management Science*, Vol. 47, 2001, pp. 864–879.
- [22] S. Zionts, “The criss-cross method for solving linear programming problems,” *Management Science*, Vol. 15, 1969, pp. 426–445.