FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Column Generation at Strip Level for the K-Staged Two-Dimensional Cutting Stock Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Franz Leberl, BSc.
Registration Number 0826666

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Dipl.-Ing. Frederico Dusberger

Vienna, 10th November, 2015

Franz Leberl

Günther Raidl

Erklärung zur Verfassung der Arbeit

Franz Leberl, BSc.
Columbusgasse 44, 1100 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. November 2015

Franz Leberl

Danksagung

An dieser Stelle möchte ich Ao. Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl für seine Hilfe und Geduld mit mir danken. Auch möchte ich Dipl.-Ing. Frederico Dusberger für seine Ratschläge und unzählige Treffen danken. Beiden danke ich für die Betreuungen dieser Arbeit.
Ich möchte mich auch bei meiner Familie bedanken, die mich durch das gesamte Studium hindurch nicht nur materiell, sondern auch emotional immer unterstützt hat.

Acknowledgements

I would like to thank Ao. Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl for his help and patience with me during this thesis. I would also like to thank Dipl.-Ing. Frederico Dusberger for his council and countless meetings with me.
I would also like to thank my family, which supported me not only financially, but also emotionally during the entirety of my studies.

Kurzfassung

Viele industrielle Anwendungen existieren für das Packen von kleineren Objekten in größere Behälter, so-genannte *Bins*, sodass die Anzahl der verwendeten Bins minimal ist. Beispiele sind die Holz, Metall oder Glass Industrien, in der die Kundenbestellungen aus größeren Stücken Lagermaterial zugeschnitten werden müssen. Wenn sowohl Bins, als auch die kleineren Objekte, so-genannte *Elements*, rechteckig sind, ist vom 2-dimensionalem *Bin Packing Problem* (2BP) bzw. dem 2-dimensionalem *Cutting Stock Problem* (2CS) die Rede. In dieser Diplomarbeit werden weiters nur so-genannte Guillotinenschnitte erlaubt, was orthogonale Schnitte von einer Seite des Bins zur anderen sind. Diese Schnitte haben zur Folge, dass man das 2CS weiters über ihre Anzahl an *Stages* definieren kann, die mit K bezeichnet wird. Ein *Stage* ist hier eine Reihe paralleler Schnitte. Das Ziel dieser Diplomarbeit ist es, eine neue Variante von *Column Generation* für das 2-staged und 3-staged 2CS zu präsentieren, die auf der Generation von so-genannten *Strips* basiert.

Zuerst wird das Problem formal definiert, und das Prinzip der Column Generation genauer erläutert. Darauf folgt eine Untersuchung der Litertur über das 2CS. Literatur zu dem *Knapsack Problem* wird auch vorgestellt, da es sich dabei um ein relevantes Subproblem der Column Generation handelt. Dies beinhaltet Varianten für das *Unbounded* als auch für das *Bounded Knapsack Problem* (UKP und BKP). Danach wird das *Stage Shifted Column Generation* (SSCG) präsentiert. Dazu gehören Definitionen für das *Master Problem* als auch für die relevanten *Pricing Probleme* für den Fall $K = 2$ und $K = 3$. Das Master Problem wird als ILP formuliert, genauer einem *Set Covering Problem*, während die Pricing Probleme Varianten eines Knapsack Problems darstellen. Für das UKP wird der effiziente algorithmus EDUK implementiert, während für das BKP ein neuer Algorithmus vorgestellt wird, der den gesamten Knapsack immer Element für Element bearbeitet, und BKP-Generator heißt. Für $K = 3$ wird BKP-Generator adaptiert, und DP-Generator genannt. Zwei Heuristiken, die auf der Randomisierung von DP-Generator beruhen, werden ebenfalls vorgestellt. Es wird außerdem eine *Integrality Heuristic* vorgestellt, da die Ergebnisse des Master Problems meist nicht integral sind. Die verschiedenen Pricing Probleme werden experimentell miteinander, und mit einer *Insertion Heuristic* als auch einer *Dynamic Programming* Implementation verglichen. Die Resultate zeigen gute Lösungen für die LP Relaxierung, und auch die integralen Lösungen können teils besser als die vorherigen Implementationen sein.

Abstract

Several industrial applications exist for packing non overlapping objects, called *elements*, into larger objects, called *bins*, such that the total number of used bins is minimal. Examples are wood, metal and glass industries, where the customers' orders must be cut from larger pieces of stock material. Particularly when high volumes of stock material are used, even small improvements can directly increase profitability. Assuming both elements and bins are rectangular, the problem is called the 2-dimensional bin packing problem (2BP) or cutting stock problem (2CS), both of which are NP-hard. Only guillotine cuts are allowed, which are orthogonal cuts from one side to another. This leads to a further specification of the 2CS, which is the number of *stages* it allows, denoted by K. A stage is a series of parallel cuts. The aim of this thesis is to present an efficient implementation of a new strip-based column generation approach for the 2-staged and 3-staged 2CS with guillotine cuts.
First, the problem and column generation are introduced and formally defined. This is followed by a study of relevant literature concerning the 2CS and 2BP. Literature for the knapsack problem is also studied, as this is a relevant subproblem of column generation. This includes variants of dynamic programming both for the *unbounded* and *bounded knapsack problem* (UKP and BKP). After that, the *Stage Shifted Column Generation* (SSCG) is presented. This entails both definitions for the *master problem* and the relevant *pricing problems* for $K = 2$ and $K = 3$. The master problem is formulated as a set covering problem, while the pricing problems are variants of knapsack problems. EDUK is an efficient implementation for the UKP, while a new algorithm is developed for the BKP, called BKP-Generator. It essentially processes the entire knapsack element by element. For $K = 3$, BKP-Generator is adapted, and called DP-Generator. Two heuristic algorithms are also presented, which rely on randomizing DP-Generator. Finally, an integrality heuristic is introduced, as the master problem offers possibly fractional results. The different pricing problems are experimentally tested and compared to an insertion heuristic and a dynamic programming implementation. The results show good LP relaxed solutions, and integral results are competitive with previous implementations.

Contents

CHAPTER 1

Introduction

The two-dimensional bin packing problem (2BP) is an NP-hard problem, and occurs in many real world industries. It consists of packing objects of different sizes into a finite number of bins without overlap, minimizing the number of bins used. There are many applications, from efficiently packing transportation vehicles or storage containers to designing integrated circuits or solving scheduling tasks. A common problem in the wood, metal, glass or paper industries is to cut standard pieces of stock material (such as metal or glass sheets) into smaller customer ordered elements, while minimizing waste. This specific kind of 2BP is referred to as the two-dimensional *cutting stock problem* (2CS). Industrial applications of cutting stock problems are particularly relevant when the stock material is produced in large rolls or sheets in high volumes and are then further cut into smaller units. Even small improvements in how the elements are cut lead to reduced waste, which can directly increase profitability by reducing wasted cutting time, energy, or wasted stock material. Due to the nature of NP-hard problems, some solution approaches are better at solving certain instances than others. This thesis presents an algorithm for solving instances with few different elements *types*, but many copies of the same element type. We first introduce the problem formally. This is followed by an explanation of column generation. The main part of the thesis focuses on the new strip-generating algorithm, followed by a chapter on the results. Finally, conclusions are drawn and future work is discussed.

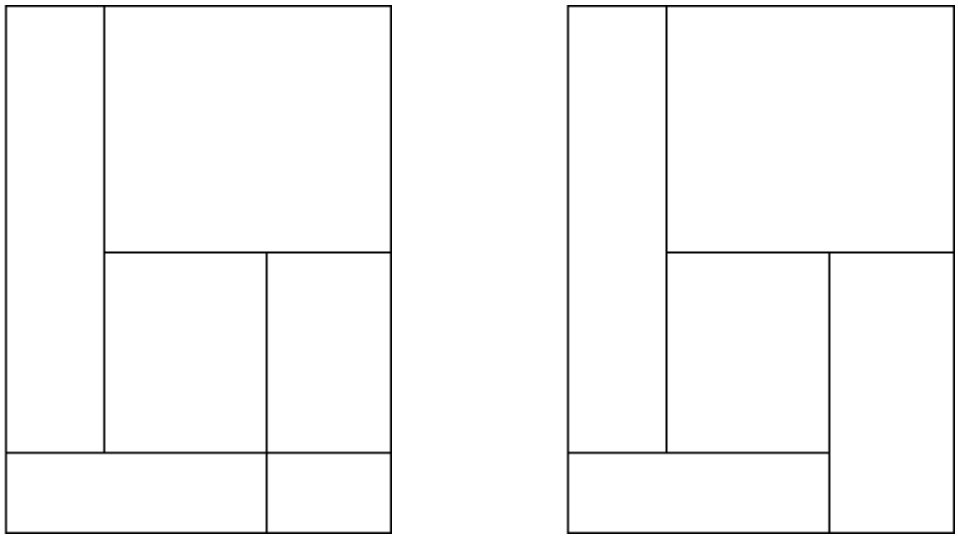
1.1 Problem Definition

The 2CS is defined as follows. Given n rectangular *element types* $E = \{1, \dots, n\}$, each $i \in E$ having height $h_i \in \mathbb{N}^+$, width $w_i \in \mathbb{N}^+$ and demand $b_i \in \mathbb{N}^+$, the objective is to cut them out from a minimal number of identical *sheets*, each with height $H > 0$ and width $W > 0$. A cutting pattern is a pattern for how elements are to be cut from the sheet, see Figure 1.4a. A special requirement on the cutting patterns is that only

orthogonal *guillotine* cuts are allowed, which means pieces are only to be cut horizontally or vertically from one border to the one opposite (see Figure 1.1). Since only guillotine cuts are allowed, a 2CS problem can be further specified by its number of *stages*. Each stage consists of a series of parallel cuts. In a K-2CS then, K is the number of times the cuts alternate between being horizontal and vertical, where w.l.o.g. the first stage is always horizontal. A *strip* is the result of the 1st stage of cuts, and a *stack* is the result of the 2nd stage of cuts. Therefore, a feasible solution to the 3-staged 2CS consists of patterns, each pattern consists of a set of strips, each strip consists of a set of stacks, and each stack consists of a set of elements of equal width. An example of a 4-staged pattern, including strips and stacks, can be seen in Figure 1.2. Typically, both sheets and elements can be rotated. The objective function Z is a function of the used sheets, minus the largest strip of waste. If all cutting patterns are in *normal form* (see section 1.1.2), and sorted by their largest unused strip, then the objective function is

$$Z = N - \frac{H - c_l}{H} \tag{1.1}$$

where c_l is the height of the last strip, and N is the number of used sheets.



(a) A cutting pattern with guillotine cuts. (b) A cutting pattern not using guillotine cuts.

Figure 1.1: Guillotine cuts

Several state-of-the-art methods for solving the 2CS are of heuristic nature employing a framework based on a set covering integer linear programming formulation. Such a model uses the set of all possible sheet patterns.

1.1.1 Restrictions

Given the basic problem definition of the K-2CS, we assume the following restrictions in the context of this thesis:

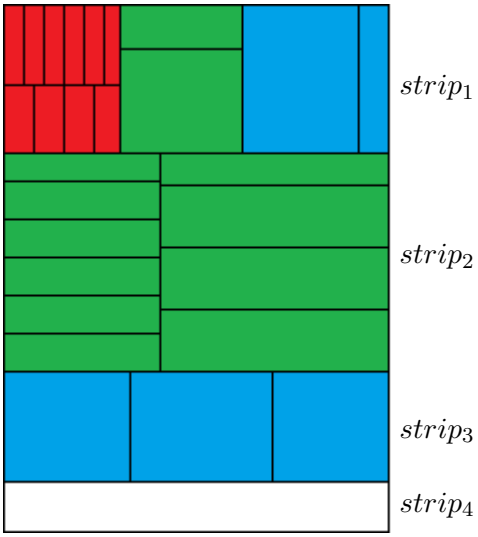


Figure 1.2: A sheet cutting pattern. The white element is done after the 1st stage of cuts, the blue elements after the 2nd stage, the green elements form a stack and are cut in the 3rd stage. The red elements require a 4th stage.

- Patterns must only contain guillotine cuts.
- The objective function is simply the number of sheets used.
- Only $K = 2$ and $K = 3$ are considered.
- Rotation of elements and sheets is ignored.

1.1.2 Solution

A solution to the 2CS is represented by a *cutting tree* (figure 1.4). A cutting tree consists of a root node, which represents all patterns, and contains all used sheet cutting patterns as *subpatterns*. Each sheet cutting pattern is then represented by a number of subpatterns, which are strips, called *horizontal compounds* (K=2). These horizontal compounds are each further represented by a number of subpatterns, which are stacks, called *vertical compounds* (K=3). Every compound always stores the contained waste. A *feasible* sheet cutting pattern is a pattern which fits into the sheet, and can be cut in the given stages using guillotine cuts. This is assured bottom up, as every compound must be feasible (must fit into a higher compound). A sheet cutting pattern is always represented in *normal form*. This is achieved by moving each element into its uppermost and leftmost position. The following must hold true:

- Waste material only appears at the bottom and right side of each stack, strip and pattern.

- The order of stacks in a strip is descending by height from left to right.

An example of normal form can be seen in figure 1.3.

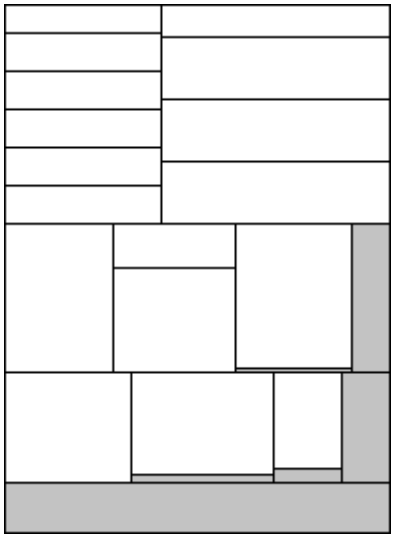
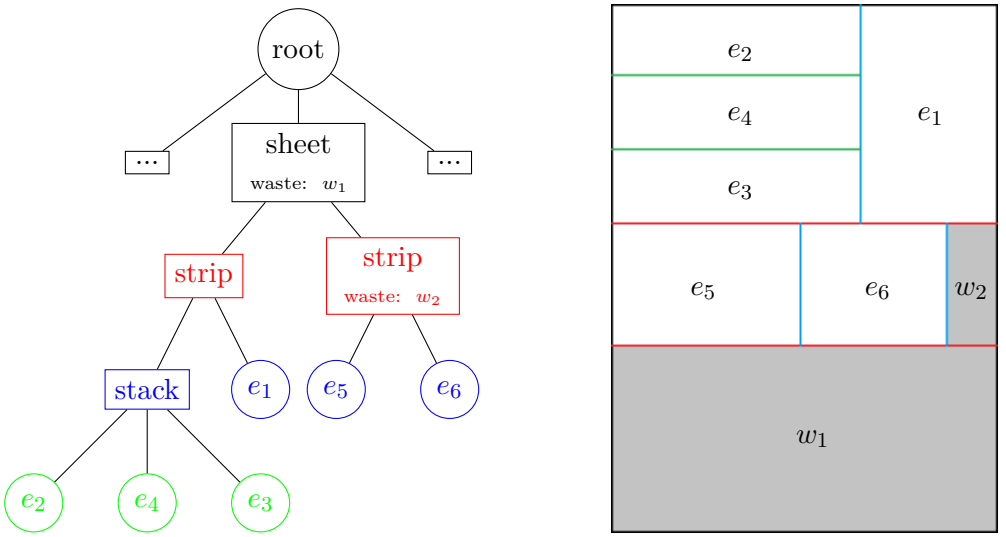


Figure 1.3: A cutting pattern in normal form. The grey area is waste.



(a) Cutting tree.

(b) Corresponding cutting pattern.

Figure 1.4: Cutting tree

1.2 Column Generation

Column generation is a technique used to efficiently solve large linear programming problems, originating from the Dantzig-Wolfe decomposition [DW60]. The general idea is that many large linear programs are too large to consider all variables. Variables that are non-basic and 0 in the optimal solution theoretically do not need to be considered when solving the problem. Column generation leverages this idea in cases where there are many of these variables, by solving the problem for a small subset of variables, and iteratively increasing the number of (promising) variables. Gilmore and Gomory [GG61] first published a paper in which they used column generation for the cutting stock problem. One big benefit of column generation is that when formulating a problem, the *master* problem (section 1.2.2) and *pricing* problem (section 1.2.3) structure allows for formulations that are very intuitive and succinct. This is especially true for the Stock Cutting Problem, in which standard Integer Linear Programming models become very large with increasing K.

1.2.1 Dantzig-Wolfe Decomposition

Dantzig-Wolfe decomposition is a standard way to decompose an integer programming model into a master problem and one or several *pricing* (sub-) problems. Assume we are given the following integer programming model:

$$z = \min \sum_{i=1}^{n} (c_i)^T x_i \tag{1.2}$$

$$\sum_{i=1}^{n} (A_{ij} x_i) = b_j \qquad \text{for } j = 1 \dots m \tag{1.3}$$

$$x_i \in X_i \qquad \text{for } i = 1 \dots n \tag{1.4}$$

where c_i is the cost vector and A_{ij} is the constraint matrix. Since we are dealing with integer subproblems, the X_i can be equivalently seen as a finite integer set:

$$X_i = \{x_{pi} | p = 1, \dots, P\}.$$

We can now alternatively write X_i as a set of convex combinations

$$X_i = \{x_i | x_i = \sum_{p=1}^{P} \lambda_p x_{pi}, \sum_{p=1}^{P} \lambda_{pi} = 1, \lambda_{pi} \in \{0, 1\} \text{ for all } \lambda_{pi}\}. \tag{1.5}$$

We can now plug the above equation (1.5) into the integer program (1.2), which means introducing $x_i \Rightarrow \sum_{p=1}^{P} \lambda_{pi} x_{pi}$ and the constraint $\sum_{p=1}^{P} \lambda_{pi} = 1$, where P is the cardinality of the corresponding integer set.

1.2.2 Master Problem

As a result of the Dantzig-Wolfe decomposition from above, and following LP relaxation, we get the *Linear Programming Master Problem* (LMP):

$$z^{LPM} = \min \sum_{i=1}^{n} \sum_{p=1}^{P} ((c_i)^T x_{pi}) \lambda_{pi} \tag{1.6}$$

$$\sum_{i=1}^{n} \sum_{p=1}^{P} (A_{ij} x_{pi}) \lambda_{pi} = b_j \qquad \text{for } j = 1 \dots m \tag{1.7}$$

$$\sum_{p=1}^{P} \lambda_{pi} = 1 \qquad \text{for } i = 1 \dots n \tag{1.8}$$

$$\lambda_{pi} \geq 0 \qquad \text{for } i = 1 \dots n \text{ and } p = 1 \dots P \tag{1.9}$$

Constraints (1.7) are called the *linking constraints*, while the constraints (1.8) are referred to as *convexity constraints*. The names originate from the fact that the linking constraints link the variables together, while the convexity constraints form a convex set (where P is the cardinality of the associated integer set when λ_{pi} is integral, i.e. $\lambda_{pi} \in \{0, 1\}$).

Suppose we have a subset of the columns, such that for each $i = 1, \dots, n$ we have at least one column from the LPM such that the solution is feasible for the LPM. Then we can define a feasible restricted linear master problem *Restricted Linear Programming Master Problem* (RLPM):

$$z^{RLPM} = \min \ \bar{c}^T \bar{\lambda} \tag{1.10}$$

$$\bar{A}\bar{\lambda} = \bar{b} \tag{1.11}$$

$$\bar{\lambda} \geq 0 \tag{1.12}$$

where the matrix $\bar{A}$ is a submatrix of the constraint matrix, $\bar{\lambda}$ denotes the restricted set of variables we chose and $\bar{c}$ is the corresponding restricted cost vector of those variables. The associated optimal solution is $\bar{\lambda}^*$, which is found using for example the Simplex method. Let (π, μ) be the corresponding dual solution (which is an optimal solution to the dual problem due to the strong duality theorem). $\bar{\lambda}^*$ is feasible also for LPM because the solution can easily be emulated: copy over the λ_p from the optimal solution of the RLPM, and set all other λ_p to zero. Let $\pi_k, k = 1, \dots, m$ be the dual variables for the linking constraints, and $\mu_i, i = 1, \dots, n$ be the dual variables for the convexity constraints, then we have

$$z^{RLPM} = \sum_{k=1}^{m} b_k \pi_k + \sum_{i=1}^{n} \mu_i \geq z^{LPM} \tag{1.13}$$

1.2.3 Pricing Problem

The *dual problem* gives a lower bound to the master problem (*primal* problem). Formulating the dual problem of the LPM in the standard way gives the *Linear Programming Pricing Problem* (LPP):

$$\max \quad \sum_{k=1}^{m} b_k \pi_k + \sum_{i=1}^{n} \mu_i \tag{1.14}$$

$$\pi^T (A_i x) + \mu_i \leq (c_i)^T x \qquad \forall x \in X_i, \text{ for } i = 1, \ldots, n \tag{1.15}$$

And therefore the corresponding reduced cost RC_i for each $x \in X_i$

$$RC_i = ((c_i)^T - \pi^T A_i)x - \mu_i \tag{1.16}$$

Note that the columns are now rows. The reduced cost gives an estimate of the improvement on the objective function an increase in the associated variable gives. In other words, an optimistic suggestion to the master problem [PSWB11]. That means we want to check for each $x \in X_i$ for $i = 1, \ldots, n$ if we have a negative reduced cost, i.e. check if

$$((c_i)^T - \pi^T A_i)x - \mu_i < 0 \tag{1.17}$$

If we find an x for which (1.17) holds, that equivalently means that (1.15) does not hold, making the dual problem infeasable, and we need to add this x to the RLPM and search for a new optimal solution $\bar{\lambda}^*$ for the new RLPM. If all constraints for the LPP are satisfied, the optimal solution to the LPP is feasible, making it, by the strong duality theorem, also the optimal solution to the LPM. Ultimately, because of the statement from [PSWB11], saying that a smaller reduced cost is probably more beneficial to the objective function, we want to find the smallest RC_i. Instead of going through each reduced cost one by one, we can instead solve n optimization problems, one for each $i = 1, \ldots, n$:

$$RC_i = \min \ ((c_i)^T - \pi^T A_i)x - \mu_i$$

where $x \in X_i$.
Because we are working with bounded sets, for each of the n optimization tasks, there are 2 possible outcomes:

$$RC_i > 0 \text{ and } RC_i \leq 0. \tag{1.18}$$

- **Case 1:** $RC_i < 0$ for some $i \in \{1, \ldots, n\}$. Then x_i^* is the optimal solution of the optimisation for this value of i. The corresponding column has a maximal negative reduced cost. This means we found a variable for which the LPP is infeasible (as the constraint is not satisfied), so we must consider it in the RLPM. We introduce

the new column $\begin{pmatrix} (c_i)^T x_i \\ A_i x_i \\ e_i \end{pmatrix}$. This leads to a new Restricted Linear Programming Master Problem, which is reoptimized by simplex.

- **Case 2:** $RC_i \geq 0$ for all $i = 1, \ldots, n$. This means the dual solution (π, μ) we get from RLPM(1.10) is also feasible for LPP(1.14). However, LPP has not been reduced, meaning by the strong duality theorem that (π, μ) is the optimal solution for LPM as well. So we can adapt (1.13) to

$$\bar{z}^{LPM} \geq \sum_{k=1}^{m} b_k \pi_k + \sum_{i=1}^{n} \mu_i = \bar{c}^T \bar{\lambda}^* = \bar{z}^{RLPM} \geq \bar{z}^{LPM}. \tag{1.19}$$

  Thus the current $\bar{\lambda}^*$ is optimal for LPM(1.6) and we can terminate.

Since $\bar{\lambda}^*$ is only optimal for LPM, we are not done yet, as this is the LP-relaxed problem. If LPM is not integral, an option is to start branching ($\rightarrow$ branch and price, see Figure 1.5). Generally, it is best to branch on the original variables. Branching on say λ_p tends to have the problem that pricing will suggest the variable again. A common form of branching in the stock cutting problem is dividing the solution space into two parts: One where element i_1 and i_2 are on the same bin, and one where they aren't. Heuristics may also be used to reach an integral solution, such as rounding the optimal LP solution, and then repairing the solution to feasibility.

1.2.4 Cutting Stock Problem

We can directly formulate the 2CS as a set covering problem, which also takes the form of a master problem in column generation:

$$\min \quad \sum_{p \in P} x_p \tag{1.20}$$

$$\sum_{p \in P} x_p A_i^p \geq b_i \qquad \forall i = 1, \ldots, n \tag{1.21}$$

$$x_p \in \{0, 1\} \qquad \forall p \in P \tag{1.22}$$

Where P is the set of all feasible sheet cutting patterns, x_p decides if we use pattern $p \in P$ and A_i^p equals the number of times pattern p contains element i. There is no associated cost to the decision variable x_p, as we want to use the minimal number of patterns, where one pattern uses one bin. The complexity of the problem lies in the size of 2^P, i.e. it is exponential in the size of P. Then we take the following steps:

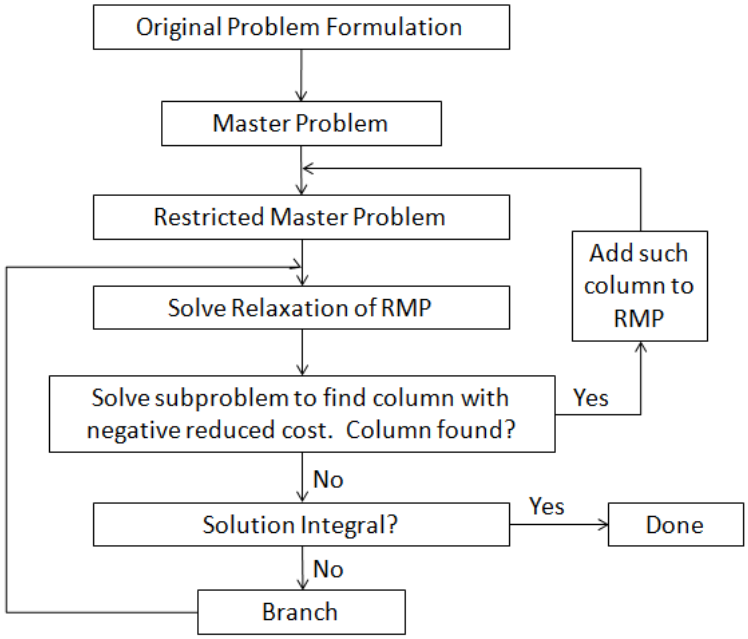


Figure 1.5: Outline of branch & price, adapted from [MAS].

- Create an initial feasible solution with a subset of patterns $P^{'} \subset P$, reducing the number of columns. The initial patterns are most commonly found using some simple heuristic, such as variants of First Fit.
- Search for a pattern yielding negative reduced cost

This formulation of the master problem leaves us with the responsibility to find good sheet cutting patterns as the pricing problem. Finding entire sheet cutting patterns can be done in any number of ways, using heuristics (first fit is popular), metaheuristics (evolutionary algorithms see[PR04]), using dynamic programming or comprehensive ILP models. The dual problem is

$$\max \quad \sum_{i=1}^{n} b_i \pi_i \tag{1.23}$$

$$\sum_{i=1}^{n} A_i^p \pi_i \leq 1 \qquad \forall p \in P \tag{1.24}$$

$$\pi \geq 0 \tag{1.25}$$

where π_i are the dual variables of the LP-relaxed master problem. The reduced cost of a pattern p is given by

$$RC_p^{\pi} = 1 - \sum_{i=1}^{n} A_i^p \pi_i \tag{1.26}$$

The goal now is to find a pattern for which the reduced cost is negative, and then add said pattern to P', and reoptimize the master problem. Note that finding a negative reduced cost is the same as checking the dual problem for feasibility. This process is iterated until no more patterns with negative reduced cost can be found. At this stage, the LP-relaxed problem has been solved optimally, given all pattern have been checked.

1.3 Motivation and aim of the work

Even small improvements in solution quality can have a large impact on production costs in industry. So although there are many algorithms for solving not only the 2CS for 2 and 3 stages, but for arbitrarily many stages, there is a demand for algorithms that solve specific kinds of instances particularly well. The aim of this thesis is to present an efficient implementation of a new strip-based column generation approach, which hopes to find improved solutions for instances of many elements, but few different element types.
The strip generating idea basically entails two things

- The restricted master problem becomes more complex. Essentially, one stage gets removed from the pricing problem, and pushed onto the master problem, i.e. the first cutting stage is considered in the master problem.
- Since the master problem becomes more complex, the pricing problem becomes easier to solve.

As the pricing problems become easier to solve, algorithms based on exact techniques such as dynamic programming become more realistic for fast implementations, and as such are more likely to yield solutions with a negative reduced cost. The now more complex master problem may still be solved efficiently given heuristically good columns, leading to better overall solutions to the LP relaxed master problem. As a disadvantage, it is expected that this new column generation variant will require more pricing iterations to finally solve the LP relaxation of the master problem. However, in a heuristic context, when the restricted set of columns is compiled from existing promising solutions, this aspect might not be that important, as the solution might be reached quickly.
The thesis first states the problem formally and introduces the reader to basic ideas and notions common in solving the 2CS. A literature review follows, describing different other algorithms, some of which will be used as comparisons to the strip-based column generation. The subsequent chapters focus on the methodology and implementation of column generation in general and specifically the algorithms for solving the 1-staged and 2-staged pricing problems. The thesis concludes with the experiments, a short summary and a chapter on future work.

CHAPTER 2

Related Work

2CS belongs to the general class of cutting and packing problems (C&P). Because of their similarity, 2CS and 2BP can almost be used interchangeably, leading literature to refer to pieces of stock as either bins or sheets, and solutions as either packings or patterns. As previously mentioned, the 2CS is an NP-hard problem, and is reducible to the knapsack problem. A topology of C&P was done by Dyckhoff [Dyc90] and later adapted by Wäscher et al.[WHS07]. A distinction is made between the *Single Stock-Size Cutting Stock Problem* (SSSCSP), having a single size of stock sheets, and the *Multiple Stock-Size Cutting Stock Problem* (MSSCSP), having multiple different sizes of stock sheets. We refer to the MSSCSP as the *Cutting Stock Problem with Variable Bin Size* (2CSV) and to the SSSCSP as the 2CS.
The 2-staged 2CS was first considered by Gilmore and Gomory in a series of papers [GG61] [GG63] [GG65], who formulated it as a set covering problem, and solved it using delayed column generation (column generation). Many state-of-the-art solutions to the 2CS, both heuristic and exact, use a set covering formulation based on their work, usually differing in the respective solution approaches to the pricing problem. Oliveira and Ferreira [OF94] presented the fast delayed column generation, which solves the 3-staged 2CS using a greedy heuristic on the pricing problem. Monaci and Toth [MT06] also create patterns using a greedy heuristic, and solve the set covering formulation heuristically. Alvarez-Valdes et al. [AVPT02] proposed a method involving GRASP or tabu search to solve the pricing problem. They also improve the results using a path-relinking algorithm, as first introduced by Glover [Glo97]. A hierarchy of different approaches to the pricing problem is used by Puchinger and Raidl [PR07] [PR04] and Puchinger et al. [PRK04]. The hierarchy consists of a fast greedy construction heuristic, an evolutionary algorithm, and finally an ILP model to generate columns. The idea is to minimize the use of the computationally more expensive methods by previously using the faster, cheaper ones. Since the pricing problem of the 2CS consists of a 2-dimensional knapsack problem, a promising method for pricing is dynamic programming. Methods including dynamic

programming are demonstrated by Cintra et al. [CMWX08] and Morabito et al.[MP10]. Cintra et al. [CMWX08] also demonstrate that both 3-staged and 4-staged sheet patterns offer similarly good results, sometimes even being optimal. This suggests the 3-staged 2CS offers sufficiently good solutions compared to unbounded many stages. This is compounded by the real world restrictions that glass cutting machines impose, often only being capable of 3 stages.

In [Cui13], Cui presents a dynamic programming algorithm for finding sheet patterns for the 3-staged 2CS. Such a problem is formally called *Single Large Object Placement Problem* (SLOPP) [WHS07], or also *Constrained Two Dimensional Cutting* (CTDC). Gilmore and Gomory [GG66] introduced two-dimensional knapsack functions and a dynamic programming approach for the SLOPP. Herz [Her72] used an exact recursive procedure using *discretization points*. Scheithauer [Sch97] further reduces the possibilities with his definition of so-called *reduced raster points*. Beasley [Bea85] built upon the algorithm by Gilmore and Gomory using the discretization points from Herz. Other approaches to the SLOPP have been proposed by Hifi et al. [HM05] and Cui et al. [CHH04] [CWL05].

A recent survey of 2-dimensional bin packing was done by Lodi et al. [LMM02], and an approach to the 2-staged 2BP was described by Lodi and Martello [LMV04]. A good overview of current construction heuristics, such as $first\text{-}fit\ decreasing\ height$ and $finite\ first\text{-}fit$, can be found in the paper by Lodi and Martello [LMV02]. A closer look is taken in the following sections. In [Fle13], Fleszar presents three insertion heuristics. The heuristics pack one element at a time, either into an existing sheet, or if that is not possible, into a new sheet, akin to the fit-decreasing and best-fit-decreasing heuristics from [MT90b], making it a *one-phase* algorithm.

Dusberger and Raidl [DR14] [DR15b] present a Variable Neighbourhood Search algorithm using Very Large Neighborhood structures based on the "ruin-and-recreate" principle. Literature on the 2CSV (multiple different sheet sizes) is rather scarce. Dusberger and Raidl [DR15a] introduce an approach for the $K\text{-}2CSV$ that scales well to instances that are large in terms of the total demand over all elements. In this approach, the patterns are computed using a construction heuristic that exploits congruencies resulting from the large number of identical elements. A meaningful selection of sheet types to be used is realized by a beam-search approach. Hong et al. [HZL$^+$14] use fast construction heuristics for sheet patterns, and uses backtracking for sheet selection.

Cui and Liu [CL08] suggest *T-shape homogenous block patterns* as a different type of pattern, as opposed to staged patterns. Here, a sheet consists of *homogenous blocks*, which consist of *homogenous strips*, which are strips containing only one element type. They propose a dynamic programming recursion to generate blocks, and solve knapsack problems to place the blocks on the sheets. *T-shape* patterns are a super set of 2-stage patterns, but not of 3-stage patterns.

A common problem in certain industries is stock with defects. Resources like wood or leather naturally come with defects such as knotholes for wood or areas of inferior quality

for leather, areas which a solution should avoid. Afsharian et al. [ANW14] propose a dynamic programming algorithm for solving the SLOPP with defects, in which the defects are approximated by rectangles.

2.0.1 One Dimensional Bin Packing

2BP is an NP-hard problem which can be approximated using greedy heuristics. Approximation algorithms are useful to understand the complexity of a problem, therefore some *greedy* approximation algorithms are presented for the 2BP. Greedy means every decision is final, i.e. no backtracking is allowed. For the validity of the approximation qualities discussed here, all heights of the elements are normalized, such that $\max_j\{h_j\} = 1$. Approximation algorithms for the 1-dimensional bin packing problem (1BP) are also used for the 2BP, so a quick overview is given here. Coffman, Garey and Johnson [CJGJ96] give an overview of approximation algorithms for the 1BP. $OPT(I)$ refers to the optimal solution of instance I.

- *Next Fit* (NF) tries to put the current element into the *current* bin. If this is not possible, the next bin is started, becoming the new current bin, and the element is placed into it. For all instances I we have $NF(I) \leq 2OPT(I) - 1$.
- *First Fit* attempts to put the current element into the first possible bin. If the element does not fit into any of the started bins, a new bin is started, and the element placed into it. For all instances I we have $FF(I) \leq \left\lceil \frac{17}{10} OPT(I) \right\rceil$ [GGJY76].
- *Best Fit* tries to add the current element to the current *best* bin, which is the bin which is filled the most, where the current element fits. If it fits in none of the bins, a new bin is started, and the element is added to it. The approximation quality is the same as for FF.
- *First Fit Decreasing* (FFD) first sorts the elements by size, and then applies FF. We have $FFD(I) \leq \frac{11}{9} OPT(I) + 4$.
- *Best Fit Decreasing* (BFD) also sort the elements by size, and then applies BF. BFD has the same approximation quality as FFD [JDU+74].

2.0.2 Two Dimensional Strip Packing

Two dimensional *strip packing* (2SP) refers to the problem of filling a bin with finite width, but infinite height, and minimizing the used height. Here, the bin is called *strip*, although this should not be confused with a strip from the 2CS. They are also *level algorithms*, which means that elements are placed into the bin from left to right, each row called a *level*. These levels are equivalent to the strips known from the 2CS with $K = 2$, with trimming allowed. This means each strip or level can contain elements of varying heights, with the strip or level height defined by the highest element. These algorithms are also used in the approximation algorithms for the 2BP, so an overview is

given. This section and the next orientate themselves by the two papers by Lodi et al. [LMV02] and Lodi et al. [LMV04].

- *Next Fit Decreasing Height* (NFDH) tries to put the current element into the current level. If this does not work, a new level is started, and the element added to it. For all instances I we have $NFDH(I) \leq 2OPT(I) + 1$ [CGJT80].
- *First Fit Decreasing Height* (FFDH) attempts to add the current element into the first level that it fits into. If it fits into no level, a new level is initialized with the current element. For all instances I we have $FFDH(I) \leq \frac{17}{10}OPT(I) + 1$ [CGJT80].
- *Best Fit Decreasing Height* (BFDH) tries to put the current element into the best level it fits. The best level is the one with the least remaining horizontal space. A new level is initialized if the element does not fit into any other levels.

2.0.3 Approximation Algorithms for the 2BP

The following approximation algorithms for the 2BP are all greedy heuristics. Moreover, the heights are normalized, as previously stated. They built upon the solutions from the 2SP and the 1BP. Two distinct families can be observed. The first are the *one-phase* algorithms, in which the elements are packed directly into the bins. In the *two-phase* algorithms, first levels are created using a 2SP algorithm, which are then placed into the bins using some 1BP algorithm.

- *Hybrid First Fit* is present by Chung et al. in [CGJ82]. It is a two phase algorithm. In the first phase the 2SP is solved using FFDH. The different levels from the resulting *strip packing* are then sorted by height, and are then used to solve the 2BP using FFD. The second phase is a 1BP. The approximation quality is given as $HFF(I) \leq \frac{17}{8}OPT(I) + 5$, however this bound is not tight.
- *Finite Best Strip* (FBS) is presented by Berkey and Wang [BW87]. It uses BFDH in the first phase, and BFD in the second phase.
- Frenk and Galambos [FG87] analyzed another variation, *Hybrid Next Fit* (HNF). It uses NFDH in the first phase, and NFD (*Next Fit Decreasing* for the 1BP) in the second phase. They show that the bound is $HNF(I) \leq 3.382\ldots OPT(I) + 9$, where $3.382\ldots$ is an approximation of a tight but irrational bound.
- Berkey and Wang [BW87] further propose the *Finite Next Fit* (FNF), which is essentially the same algorithm as HNF. Although HNF is a two-phase algorithm and FNF is a one-phase algorithm, using NFD for the second phase equates to to immediately filling the current or next, new bin, making it technically also one-phase. Both papers were published in the same year.

2.0.4 Approximation Algorithm for the UKP

Lawler presents a fully polynomial-time approximation scheme (FPTAS) for the UKP in [Law77]. It is an adaption from the KP. A FPTAS is an approximation algorithm with a full guarantee on the optimal solution within a parameter, meaning the solution can be arbitrarily close to the optimal solution, at the cost of runtime. Given the parameter ϵ, for which $(0 < \epsilon \leq 1)$ holds, the runtime is bounded by a polynomial of the input size and in a parameter $1/\epsilon$, in this case $O(n + 1/\epsilon^3)$. Let OPT be the optimal solution to an arbitrary maximization problem, and APP the solution given by the FPTAS, then the guarantee is given by equation 2.1

$$OPT - APP = \epsilon OPT. \tag{2.1}$$

This means the solution given by the FPTAS is a factor of $(1 - \epsilon)$ from the optimal solution. The fundamental idea of this approach is to only use the most efficient items, the number of which is defined by the parameter ϵ.

2.0.5 Branch & Bound Algorithm for the UKP

Branch & Bound algorithms consist of enumerating all possible solutions while updating upper and lower bounds. The lower bound is given by the current global best solution, while the upper bound is the best future solution possible in the current branch. If the upper bound is smaller or equal to the current lower bound, the current branch can be abandoned (pruned). Branch & Bound has the optimal solution when all branches are finished or pruned. The most efficient Branch & Bound algorithms for the unbounded knapsack problem [KPP04] are MTU1 and MTU2, both by Martello and Toth [MT77, MT90a], where MTU2 improves on MTU1. Both use the same calculation for the upper bound, U_2, which calculates the profit using the three most efficient items. Assuming the elements are sorted by increasing efficiency (p_i/w_i), the LP-relaxation yields the trivial bound

$$U_1 := U_{LP} = \lfloor z^{LP} \rfloor = \left\lfloor p_1 \frac{c}{w_1} \right\rfloor. \tag{2.2}$$

This can be further built upon by considering the residual capacity $c_r := c - w_1 \left\lfloor \frac{c}{w_1} \right\rfloor$ which is left over after packing the maximal number of element 1 into the knapsack. We can pack the residual capacity with element type 2, given by $\beta_2 := \lfloor c_r/w_2 \rfloor$. Now we distinguish between two cases. The first is where we pack exactly β_2 copies of element 2 into the residual capacity, and the now remaining capacity is filled with element 3. The other case is where we pack exactly $\beta_2 + 1$ copies of element 2 into the knapsack, and fix the number of copies of element 1 accordingly. This yields the upper bound used in MTU1 and MTU2 [MT90a]:

$$U_2 := max \left\{ \left\lfloor p_1 \left\lfloor \frac{c}{w_1} \right\rfloor + p_2\beta_2 + (c_r - w_2\beta_2)\frac{p_3}{w_3} \right\rfloor, \right. \tag{2.3}$$

$$\left\lfloor p_2(\beta_2+1) + p_1\beta_1 + (c - w_2(\beta_2+1) - w_1\beta_1)\frac{p_2}{w_2} \right\rfloor \Bigg\}.$$

where $\beta_1 := \left\lfloor \frac{c-w_2(\beta_2+1)}{w_1} \right\rfloor$, describing the amount of copies of element 1 can be used in the second case. After sorting the elements by descending efficiency (p_1/w_i), the recursive MTU1 algorithm proceeds roughly as follows:

- Fill the current knapsack with the current most efficient item.
- Update lower and upper bounds.
- Check for optimality or pruning of the current branch.
- Call MTU1 with the next element as starting element and the new capacity.

Once the knapsack is full, MTU1 proceeds to exchange the previously more efficient elements with less efficient ones, updating the upper bound at each replacement, thereby covering the entire search space. MTU2 builds upon MTU1 by using the *core concept*. The idea of the core concept is that elements with high efficiency are more likely to be included in an optimal solution than those with lower efficiency. Therefore, MTU1 only needs to be executed with a *core* of relevant items. When that is done, elements can be added to the core, and the process can be repeated. MTU2 is done once the upper bound equals the current solution, or all elements have been added to the core or have been eliminated. To eliminate an item, MTU2 adds one copy of an element not in the core and calculates the upper bound with the remaining knapsack. If that upper bound is worse than the current solution, that element can be eliminated.

2.0.6 Transformation of BKP to KP

One popular approach to the BKP is to define an appropriate binary knapsack problem (KP) which consists of one distinct element for every copy of every element type in the BKP. Since every element type i gives rise to b_i items, the resulting KP has $\sum_{i=1}^{n} b_i$ items. A more efficient approach involving coding the number of copies of every element type i in a binary format can be found in [KPP04], resulting in a KP of size $n + \sum_{i=1}^{n} \lfloor \log b_i \rfloor$. Given the upper bound for b_i from 3.14, this leaves the number of elements in the transformed KP instance in the order of $O(n \log c)$. This means that especially in the case of large capacities, as is usually the case in our instances, this is seems to be an unfavorable technique.

2.0.7 Approximation Algorithm for the BKP

$FPTAS$ for the BKP have been suggested by transforming the problem into a KP. This results with the capacity present in the worst case time complexity, something we wish to avoid. Dedicated $FPTAS$ for the BKP are not present in literature [KPP04] [MJ07]

however. A $FPTAS$ for the Bounded Set-up Knapsack problem (BSKP) is presented in [MJ07] with a runtime of $O(n * 1/e^2)$, which can be adapted to the BKP. A BSKP is a generalisation of the BKP in which each element type is associated with a *set-up weight* and *set-up value*. These values are applied to the objective function if any copies of that element type are in the knapsack. In [KPP04] a $FPTAS$ based on the $FPKP$ algorithm in [KP04] is suggested, resulting in a time complexity of $O(n * 1/\epsilon^4 + n \log n * 1/\epsilon^3)$ and a space complexity of $O(n * 1/e^3)$, the currently best [MJ07]. The idea behind this $FPTAS$ is to first greedily calculate a lower bound z^l of the objective function. Then the elements are split into small and large items, with all element types satisfying $p_i \leq z^l\epsilon$ being small items. Dynamic programming is then applied to the large element types. Using this solution, for each candidate profit value we check the corresponding dynamic programming function, and greedily fill the remaining capacity with small items.

2.0.8 Branch & Bound Algorithm for the BKP

According to [KPP04], research into B&B algorithms halted because of the observation that B&B applications for the transformed KP (section 2.0.6) outperformed B&B algorithms customized for the BKP. The most promising Branch & Bound algorithms for the bounded knapsack problem [KPP04] then are MTB1 and MTB2, close relatives of MTU1 and MTU2 described in section 2.0.5 both by Martello and Toth [MT77, MT90b], where MTB2 improves on MTB1. Both use the same calculation for the upper bound, U_2, which calculates the profit similarly to the unbounded case. Assume the element types are sorted by decreasing efficiencies. The LP-relaxation yields the trivial bound

$$U_{LP} = \lfloor z^{LP} \rfloor = \left\lfloor \sum_{j=1}^{s-1} p_j b_j + \left(c - \sum_{j=1}^{s-1} w_j b_j \right) \frac{p_s}{w_s} \right\rfloor . \tag{2.4}$$

Considering the residual capacity $c_r := c - \sum_{j=1}^{s-1} w_j b_j$ which is left over after greedily packing elements of types 1 to $s-1$ into the knapsack, there are again two options. We can either pack exactly $\beta_s = \left\lfloor \frac{c_r}{w_s} \right\rfloor$ into the knapsack, and the remaining capacity is filled with element type s+1. In the other case, $\beta_s + 1$ elements of type s are put into the knapsack, reducing the number of elements $s-1$. This yields the upper bound used in MTB1 and MTB2 [MT77]:

$$U_2 := max \left\{ \left\lfloor \sum_{j=1}^{s-1} p_j b_j + p_s \beta_s + (c_r - w_s \beta_s) \frac{p_{s+1}}{w_{s+1}} \right\rfloor , \right. \tag{2.5}$$

$$\left. \left\lfloor \sum_{j=1}^{s-1} p_j b_j + p_s(\beta_s + 1) + (c_r - w_s(\beta_s + 1)) \frac{p_{s-1}}{w_{s-1}} \right\rfloor \right\} .$$

The recursive MTB1 algorithm proceeds roughly as follows:

- Fill the current knapsack with the current most efficient element (while the demand lasts).
- Update lower and upper bounds.
- Check for optimality or pruning of the current branch.
- Call MTB1 with the next element as starting element and the new capacity.

Once the knapsack is full, MTB1 proceeds to exchange the previously more efficient elements with less efficient ones (branching), updating the upper bound at each replacement, thereby covering the entire search space. This is identical to the case of the UKP, with the exception that the current most efficient element can only be added while the demand lasts, i.e. exchanging elements can happen before the knapsack is full. MTB2 builds upon MTB1 by using the *core concept*, identical as in section 2.0.5. As a matter of fact, MTB2 transforms the BKP into a KP. Although claimed possible [KPP04], directly applying MT2 (the underlying algorithm for the KP) on the BKP was not done in the publication.

CHAPTER 3

Stage Shifted Column Generation

The fundamental idea behind the algorithm *Stage Shifted Column Generation* (SSCG), compared to standard column generation implementations, is to shift the computational burden of one stage from the pricing problem onto the master problem. The name for the algorithm is therefore chosen to reflect the shifted complexity of both the master and pricing problem. In other words, this means that the pricing problem generates a new strip with each successful iteration. This is in contrast to the standard column generation, where entire cutting patterns are generated.

3.1 Insertion Heuristic

Column generation requires an initial set of columns from an initial feasible solution. For the SSCG, this is achieved by use of an insertion heuristic. In [Fle13], Fleszar presents three insertion heuristics. Dusberger and Raidl [DR15a] present an insertion heuristic based on the critical-fit insertion heuristic used by Fleszar. This insertion heuristic is used to find a preliminary feasible solution for the master problem. An outline is described below:

1. Order all element types by decreasing area and let $b_i^r \in \mathbb{N}$ be the *residual demand* of element type i.
2. Determine the *undominated* element types for which there is a demand, i.e. $b_i^r > 0$. An element type $i \in E$ *dominates* another type $j \in E$ if $w_i \geq w_j$ and $h_i \geq h_j$.
3. Determine the element i_c from the undominated element types which fits in the least amount of sheets.
4. Either insert i_c in the best sheet possible, or insert it in a new sheet. Reduce the remaining demand, i.e. reduce $b_{i_c}^r$.

5. If $\exists b_i > 0$ go to step 2.

A further consideration is made in regards to the insertion. Given the structure of the cutting tree (see section 1.1.2), there are multiple types of insertions into the cutting tree, which need to be differentiated. When inserting an element into a compound c, two possibilities can be considered:

- Inserting the element as a sub-node of c
- Inserting the element in parallel to c's subpatterns.

Finally, in order to consider inserting multiple instances of one element type, a fitness criterion is used which depends on the vertical and horizontal slack after the insertion of a number of elements. The optimal quantity and position for inserting the elements is then determined by maximizing the fitness function.

3.2 Master Problem

The master problem for the SSCG is formulated as an integer linear program, similarly to the set covering formulation described in section 1.2.4. Since this model does not restrict the types of strips used, both the 2-staged and 3-staged 2CS can use the same master problem. The model requires an upper bound n of the number of sheets used, which is provided by a preliminary run of an insertion heuristic 3.1.

Let $\mathcal{P}$ be the set of all feasible cutting patterns for a strip. For each strip pattern $p \in \mathcal{P}$ and element type $i \in E$, let constants $a_{pi} \in N_0^+$ denote how many instances of element type i are contained in pattern p and constant h_p denote the height of pattern p.

Let variables $y_j \in \{0, 1\}$, $j = 1, \ldots, n$ indicate if sheet j is used, and variables $x_{pj} \in \mathbb{N}_0^+$, $p \in \mathcal{P},\ j = 1, \ldots, n$ indicate the number of strips of type p assigned to sheet j.

We can then define the *master problem* as follows:

$$\min \quad \sum_{j=1}^{n} y_j \tag{3.1}$$

$$\text{s.t.} \quad \sum_{p \in \mathcal{P}} \sum_{j=1}^{n} a_{pi} x_{pj} = b_i \qquad \forall i \in E \tag{3.2}$$

$$\sum_{p \in \mathcal{P}} h_p x_{pj} \leq H y_j \qquad \forall j = 1, \ldots, n \tag{3.3}$$

$$\sum_{p \in \mathcal{P}} h_p x_{pj} \leq \sum_{p \in \mathcal{P}} h_p x_{pj-1} \qquad \forall j = 2, \ldots, n \tag{3.4}$$

Each element must be packed according to its demand b_i (3.2). Inequalities (3.3) state that the total height of the strips packed in each sheet j must not exceed H. Furthermore,

these constraints also enforce the linking between the x- and y-variables. Inequalities (3.4) are symmetry breaking constraints, which are not necessary for the validity of the model, but strengthen it by enforcing the order of the sheets in a cutting pattern according to the normal form. The pseudo code for the master problem used in the SSCP can be seen in algorithm 3.1.

Algorithm 3.1: Master Problem

```
   Input: 2CS instance file
   Output: 2CS solution
 1 readInstance;
 2 solution = insertionHeurisict.solve();
 3 model = buildILP(solution);
 4 model.solve();
 5 duals = model.getDuals();
 6 pricing = new pricingProblem();
 7 while pricing.solve(duals) do
 8 |   newColumn = pricing.getSolution();
 9 |   model.add(newColumn);
10 |   model.solve();
11 |   duals = model.getDuals();
12 end
13 integralityHeuristic();
14 return model.getSolution();
```

3.2.1 Pricing Problem

The pricing problem for both the 2-staged and 3-staged 2CS can be formally described as follows. Let μ_i, π_j and $\varphi_{j,j'}$ be the dual variables corresponding to constraints (3.2), (3.3) and (3.4), respectively.

We do not want explicitly consider all variables x_{pj}, $p \in \mathcal{P},\ j = 1, \ldots, n$ as there are exponentially many of them. Therefore we only start with few and price in more via delayed column generation. Variable x_{pj} has reduced costs

$$\overline{c}_{pj} = -\sum_{i \in E} a_{pi}\mu_i + \pi_j h_p + \varphi_{j(j-1)} h_p - \varphi_{(j-1)j} h_p \tag{3.5}$$

$$= -\sum_{i \in E} a_{pi} + \gamma_j \cdot h_p \tag{3.6}$$

with

$$\gamma_j = \pi_j + \varphi_{j(j-1)} - \varphi_{(j-1)j}. \tag{3.7}$$

The pricing subproblem consists of finding a variable $x_{p^*j^*}$, i.e., a strip pattern $p^* \in \mathcal{P}$ for packing in a sheet $j^* \in \{1 \ldots, n\}$ having smallest reduced cost $\overline{c}_{p^*j^*}$. If $\overline{c}_{p^*j^*}$ are

negative, the variable is added to the reduced master problem, which is then resolved; otherwise, the column generation can stop as the LP-relaxation of the master problem has been solved to optimality.

The sheet j is only found in the second term of (3.6), in γ_j, and independent of the pattern p, the sheet leading to the lowest reduced costs always is

$$j^* = \text{argmin}_{j=1,\dots,n}\gamma_j. \tag{3.8}$$

Let variables $a_i \geq 0$, $i \in E$, indicate the number of elements of type i that are packed in the strip pattern p and $h(p) > 0$ denote the height of the strip pattern. The *pricing problem* can then be formulated as

$$\begin{aligned}
\max \quad & \sum_{i \in E} \mu_i a_i - \gamma_{j^*} h(p) && && (3.9)\\
& (a, h(p)) \in (K-1)\text{-2KP} && && (3.10)\\
& a_i \geq 0 && \forall i \in E && (3.11)\\
& h(p) \geq 0 && && (3.12)
\end{aligned}$$

$(K-1)$-2KP refers to the set of feasible solutions of the $(K-1)$-*stage two-dimensional knapsack problem*, i.e., all feasible $(K-1)$-stage patterns of maximum width W and maximum height H with the first stage being vertical cuts, expressed here by the vector a of numbers of elements packed from each element type and the pattern height $h(p)$.

According to the objective function (3.9), each element type has associated a profit μ_i and we are looking for a strip pattern that maximizes the total profit of all packed elements reduced by a linear "penalty" $\gamma_{j*}h(p)$ for the strip's height. Thus, we have to deal here with a special variant of the $(K-1)$-2KP.

In more detail, for $(K-1)$-2KP we may either consider the unbounded knapsack problem variant, where from each element type an arbitrary number of elements may be packed in principle, or the bounded variant, were the number of elements is limited to the demands b_i, $i \in E$. Both variants are valid and will in the end yield the same LP-bound for the master problem. On the one hand, the bounded knapsack problem might occasionally avoid less meaningful patterns, may save pricing iterations, and may yield less fractional solutions. On the other hand, the unbounded knapsack problem is typically easier to solve.

3.3 The 2-Staged 2CS

We will first study the $K = 2$ case. Feasible patterns are strips of maximum width W, where only elements of the same height h_p are placed from left to right next to each other. This means patterns that require *trimming* are not allowed. Trimming in this case refers to using elements of lower height than h_p, which would require their excess material to be trimmed.

Let $\{E_1, \ldots, E_\eta\}$ be the partitioning of all element types from E into the subsets of element types having the same height h_l, i.e., $E_1 \cup \ldots \cup E_\eta = E$, $\forall l = 1, \ldots, \eta,\ i \in E_l : h_i = h_l$, $\forall l = 1, \ldots, \eta - 1,\ l' = l + 1, \ldots, \eta,\ i \in E_l,\ i' \in E_{l'} : h_i \neq h_{i'}$.

We can then approach the pricing problem by solving an individual classical one-dimensional knapsack problem for each E_l, $l = 1, \ldots, \eta$ with element profits μ_i, element weights w_i, and maximum capacity W. From all those η knapsack solutions, we finally take the one that maximizes the objective function (3.9) considering also the pattern height h_l.

3.3.1 Unbounded Knapsack

In instances where there are more elements of each element type than can fit in the width of a strip, a simplification is to consider the knapsacks in the pricing problem as unbounded knapsack problems (UKPs). This means that each element type in N can be used unrestrictedly, even if more elements are placed into the knapsack than there is demand. This should rarely be the case in the previously mentioned instances. Although the UKP is still an NP hard problem, there exist efficient methods to solve it optimally. One such approach, a special purpose dynamic programming (DP) solution is claimed to be the most efficient solution [KPP04] to the UKP, and is explained in section 3.3.4. It exploits the *dominance* relations of elements to solve the UKP, and is called "Efficient Dynamic Programming for the Unbounded Knapsack Problem" (EDUK). Other approaches include an approximation algorithm as seen in section 2.0.4 and a Branch & Bound algorithm investigated in section 2.0.5.

3.3.2 Dominance

If two elements i and j are compared pairwise, and element i is at most as profitable and at least as heavy as element j, then element i is never used as it is always better (or at least not worse) to replace it by one or more copies of j without decreasing the total profit. This *simple dominance* was observed as early as 1963 [GG63], and can be seen in Figure 3.1a. Element j simply dominates element i since $w_j \leq w_i$ and $p_j \geq p_i$, where w_i is the weight of element i and p_i is the profit of element i.

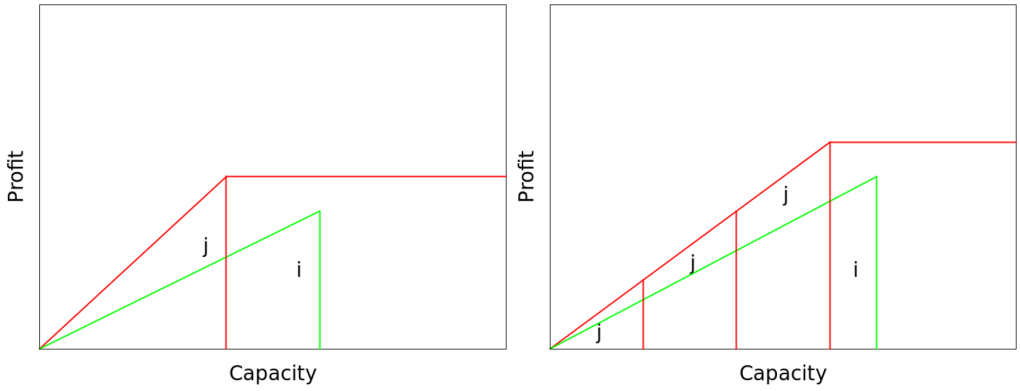
Further extensions of dominance can be found in literature [APR00]. An element i is said to be *multiply* dominated by an element j if $\lfloor w_i/w_j \rfloor \geq p_i/p_j$, meaning it is always better to replace one copy of element i by $\lfloor w_i/w_j \rfloor$ copies of element j. In Figure 3.1b the three copies of element j dominate one copy of element i.

Another dominance relation is called *threshold* dominance. An element i is threshold dominated by a set of elements J if for $\alpha \in \mathbb{N}$ and $y \in \mathbb{N}^{|J|}$ we have $\alpha w_i \geq \sum_{j \in J} y_j w_j$ and $\alpha p_i \leq \sum_{j \in J} y_j p_j$, meaning it is always better to replace α copies of element i by some combination of elements of the set J. The case where only one copy is dominated, $\alpha = 1$ is called *collective* dominance. Both collective and threshold dominance can be seen in Figure 3.1c and Figure 3.1d respectively.

In [KPP04] the following proposition can be found:

Proposition 1 *For every instance of UKP there always exists an optimal solution not containing any simply, multiply or collectively dominated element types.*

That means all simply, multiply or collectively dominated element types can be safely discarded without affecting the optimal solution, reducing the search space. Since elements that are only threshold dominated might be used in an optimal solution, they must be considered in the solution vector until the calculation reaches a capacity of the threshold dominance, at which point the element can be discarded.



(a) Simple Dominance: element j simply dominates element i.

(b) Multiple Dominance: element j multiply dominates element i.

(c) Collective Dominance: element set $\{j, k\}$ collectively dominates element i.

(d) Threshold Dominance: element set $\{j, k\}$ threshold dominates element i.

Figure 3.1: Dominance relations.

3.3.3 Periodicity

Periodicity is a property specific to the UKP. The special structure of unbounded knapsack problems, the so called *periodicity* for increasing capacity values, was detected

by Gilmore and Gomory as early as 1966, and shown in [KPP04]. It states that for capacities larger than some capacity C^*, the remaining knapsack is only filled with the most efficient element b, where efficiency is defined as p_b/w_b. The equation used in dynamic programming is then

$$z(C) = z(C^*) + \left\lceil \frac{C - C^*}{w_b} \right\rceil p_b \tag{3.13}$$

where $z(C)$, the *solution vector*, gives the optimal solution (i.e. maximum profit) for the capacity C.

3.3.4 EDUK-Generator

As previously mentioned, EDUK (Efficient Dynamic Programming for the Unbounded Knapsack Problem) is considered the most efficient algorithm for the UKP [KPP04], and as such was chosen for the UKP approach to the 2-staged pricing problem. Our implementation of EDUK, called EDUK-Generator is shown in algorithm 3.2. EDUK consists of a *reduction* phase and a *standard* phase.

- In the reduction phase, simply, multiply and collectively dominated elements are filtered out, and all other elements are added to the list F of undominated items. Since it is a dynamic programming implementation, it also calculates the optimal solutions up to the capacity of the heaviest item.
- The standard phase is a dynamic programming implementation that also considers threshold dominance and periodicity. Once there is only 1 undominated element left, we know we have achieved periodicity and the remaining knapsack can be filled with the most efficient element (i.e. the last undominated item).

Here $z[d]$ is the solution vector, storing the value of the optimal solution for a knapsack of capacity d. The helper array $l[i]$ is used for threshold dominance detection. It stores the last capacity at which element i was used in an optimal solution, i.e. $l[i] = d$ iff $z[d - w_i] + p_i > z[d]$.
The idea of slices is introduced as an optimization to reduce the calls to the threshold dominance function. Another optimization in the implementation is the idea of discretization points [Her72]. Since many knapsacks are very large with elements of similarly large weights, instead of going through each capacity, it is more efficient to calculate all the different possible linear combinations of weights of the items, and then only calculate knapsacks of those sizes. Similar implementations of these two optimization approaches are used in the original publication of EDUK[APR00].

The two loops on lines 4 and 5 create "slices", meant to reduce the calls of the threshold check on line 20. Lines 6 to 14 can be seen as a standard implementation of dynamic programming, only using elements from the list F and going from the previous item's

Algorithm 3.2: EDUK-Generator

```
1  getDiscretizationPoints;
2  sort elements in ascending order by weight;
3  F = ∅;
   // reduction phase
4  foreach item slice do
5      foreach item j in current slice do
6          foreach capacity d in discretizationPoints ∈ (w_{j-1}, w_j] do
7              z[d] := 0;
8              foreach item f in F do
9                  if z[d − w_f] + p_f > z[d] then
10                     l[f] := d;
11                     z[d] := d[d − w_f] + p_f;
12                 end
13             end
14         end
15         if z[w_j] ≤ p_j then // j not dominated by F
16             z[w_j] := p_j;
17             F := F ∪ {j};
18         end
19     end
20     checkThresholdDominance;
21 end
   // standard phase
22 foreach capacity slice in discretizationPoints do
23     if |F| = 1 then
24         periodicity;
25     end
26     foreach capacity d in current slice do
27         z[d] := 0;
28         foreach item f in F do
29             if z[d − w_f] + p_f > z[d] then
30                 l[f] := d;
31                 z[d] := z[d − w_f] + p_f;
32             end
33         end
34     end
35     checkThresholdDominance;
36 end
37 retrieveSolution;
```

capacity (excluding) to the current item's capacity (including), only using the discretization points. Line 10 stores the fact that element f was used at the current capacity d. On line 15, $z[w_j]$ has been optimally calculated without using the current element j, which of course perfectly fits into the knapsack of size w_j. If $p_j \geq z[w_j]$ that means that element j is not dominated by a collection of smaller items, therefore adding it to F. The standard phase is essentially the same as the reduction phase, just that all capacities from the discretization points are done, and no elements are added to F. Since the reduction phase solved the knapsack problem up to the heaviest item, the standard phase can start with that capacity. If periodicity is detected in line 23, the algorithm simply fills the rest of the knapsack with the remaining element (which is also the overall most efficient item), otherwise it finishes the knapsack using the dynamic programming implementation. For retrieving the solution, it is required so store the solution for each discretization point, and store the currently best discretization point.

Algorithm 3.3: checkThresholdDominance

```
1 d :=last capacity calculated;
2 foreach f in F do
3 |   if l[f] < d − w_f then
4 |   |   remove f from F;
5 |   end
6 end
```

The threshold dominance shown in Algorithm 3.3 check makes use of the vector l which is defined above. For a capacity d which has been optimally calculated, it takes all formerly undominated elements and checks if the distance in capacity between the last use and the current optimal solution is greater than w_f. If that's the case, clearly another element or combination of elements dominated element f, making it dominated and removing it from F. This algorithm is essential in reaching periodicity, and also reduces the run time of both the reduction phase and standard phase, if an element can be removed from F. However, since checking at each capacity might waste computation time, slices are introduced to control the number of calls to checkThresholdDominance. The worst case run time for Algorithm 3.3 is $O(n)$.

Discretization Points

Since only integer weights and capacities can be used with a dynamic programming implementation, it is often the case that capacities and weights are unnecessarily large, because they are simply multiplied by an integer to make all weights and capacities integral. Therefore, only looking at the *discretization points*, which are just the linear combinations of the weights of the items, can dramatically reduce runtime in practice. Algorithm DPEE (Discretization Points by Explicit Enumeration)3.4 generates the discretization points naively by creating all integer conic combinations of element heights, i.e. by explicit enumeration. In line 8 the algorithm goes through all points (= weights)

added in the previous run, starting with 0. If the points plus the current item's weight w_i fit into the knapsack in line 9, the new point is added in line 10 to *points*, and later to *discretizationPoints*. This is done from 0 to the capacity C, for all items. The worst case runtime for algorithm 3.4 is therefore $O(C\delta)$, where δ is the number of conic combinations of element heights.

Algorithm 3.4: DPEE

```
1  changed = true;
2  discretizationPoints = ∅;
3  oldPoints = 0;
4  while changed do
5  |  changed = false;
6  |  points = ∅;
7  |  foreach item i in N do
8  |  |  foreach point p in oldPoints do
9  |  |  |  if p + w_i < C then
10 |  |  |  |  add (p + w_i) to points;
11 |  |  |  |  changed = true;
12 |  |  |  end
13 |  |  end
14 |  end
15 |  add points to discretizationPoints;
16 |  oldPoints = points;
17 end
18 reduceDP;
```

To avoid a potential enumeration taking exponential time, Cintra et al. [CMWX08] use what they call DDP (Discretization Using Dynamic Programming). Our implementation can be seen in algorithm 3.5. The idea behind this algorithm is to set the profits p_i for each element $i = 1, \ldots, n$ to match its respective weight w_i, such that $p_i = w_i$. What follows is solving a knapsack problem of capacity C using dynamic programming in the standard way. Since all elements have the same efficiency of $p_i/w_i = 1$, any capacity d for which $z[d] = d$ holds is then a discretization point.

A final optimization in regards to the discretization points is a reduction of the points, as demonstrated by Scheithauer [Sch97]. The idea he proposes is that not every discretization point needs to be considered. More precisely, he removes those discretization points which separate themselves from another only by *unusable* knapsack capacity. In this sense, unusable knapsack capacity means a capacity too small for an improvement to take place. Let us demonstrate this by the use of a small example. Let $n = 3$ with $w_1 = 3$, $w_2 = 7$ and $w_3 = 8$, and let $C = 10$. The set of discretization points then is $\mathcal{P} = \{0, 3, 6, 7, 8, 9, 10\}$. Note, however, that the points $\{8, 9, 10\}$ are all "final", in the sense that none are used as the basis of an improvement, and as such are in competition

Algorithm 3.5: getDiscretizationPoints

```
1  discretizationPoints = ∅;
2  foreach capacity d in C do
3  |  z[d] = 0;
4  end
5  foreach item i in N do
6  |  foreach capacity d in C do
7  |  |  if z[d] < z[d − w_i] + p_i then
8  |  |  |  z[d] = z[d − w_i] + p_i;
9  |  |  end
10 |  end
11 end
12 foreach capacity d in C do
13 |  if z[d] = d then
14 |  |  discretizationPoints = discretizationPoints ∪ {d};
15 |  end
16 end
17 reduceDP;
```

Algorithm 3.6: reduceDP

```
1  reducedPoints = ∅;
2  dp = discretizationPoints.end();
3  foreach point p in discretizationPoints do
4  |  lim = sheetWidth − p;
5  |  while lim < dp && dp != discretizationPoints.begin() do
6  |  |  dp--;
7  |  end
8  |  reducedPoints = reducedPoints ∪ {dp};
9  end
```

with each other for the optimal solution. In other words, both points 8 and 9 are solutions with an unused capacity, which given the items, is *unusable*. Points 6 and 7 share a similar relation, since they are both the basis only to adding $i = 1$ to the solution. Scheithauer now suggests removing the points with the unusable capacity, since their solution can be stored in the respectively larger capacities. Our implementation can be seen in algorithm 3.6.

As a consequence of reducing the discretization points, a point can no longer be directly retrieved. Instead, the nearest point can be efficiently looked up, using the algorithm findNextPoint 3.7. The run time of this algorithm is $O(\log(C))$, increasing the overall worst case. In practice however, the reduction in discretization points is well worth

Algorithm 3.7: findNextPoint

Input: the int *point* to which we want to find the closest reduced point from the vector *reducedPoints*
Output: The reduced point for the given *point*

```
1  left = 0;
2  right = reducedPoints.size() − 1;
3  while left < right do
4  |  middle = (left + right)/2;
5  |  if point < reducedPoints[middle] then
6  |  |  right = middle − 1;
7  |  else
8  |  |  left = middle + 1;
9  |  end
10 end
11 return reducedPoints[l − 1];
```

the increased look up computation time. Although this algorithm is always used in conjunction with reduced points, its application is omitted from the pseudo code here for better readability.

Parameters

The parameters of Algorithm 3.2 consist of

- t - the size of element slices in the reduction phase
- q - the size of capacity slices in the standard phase

At the end of each slice, threshold dominance is tested. If the slices are too small, dominance is tested too often. However, if the slices are too large, the list F of undominated elements might remain unnecessarily large for too long. In practice this means increasing the parameter size in case the knapsack capacity C is large in comparison to the number of elements n, and decreasing the parameter size if the opposite is the case.

Complexity

The run time of our EDUK-Generator depends both on the capacity C and the number of elements n, and on algorithm 3.5, with a run time of $O(Cn)$. Because the number of executions of Algorithm 3.3 is limited by the parameters, its contribution to the worst case run time is only $O(qn + tn)$. The entire algorithm can be seen as computing a $C \times n$ table, and as such the worst case run time is $O(Cn)$, since C is obviously larger than both q and t. This is only the case when *all* points are discretization points, and periodicity is never reached. Moreover, since we are using reduced discretization points, each look up

requires an additional $O(\log(C))$ time, resulting in an overall run time of $O(Cn\log(C))$. As a pricing problem, this algorithm will be run for every available strip height, i.e. discretization point in the sheet height, giving the overall pricing problem per execution a run time of $O(WHn\log(W))$, where H is the sheet height, and W is the sheet width. Since the solution vector is two dimensional, the space requirement is $O(Cn)$, although the solution can also be reconstructed by backtracking a one dimensional solution vector, requiring $O(Cn)$ additional time.

3.3.5 Bounded Knapsack

In contrast to the UKP discussed in section 3.3.1, when the demands of the element types b_i in E play a role, the knapsacks in the pricing problem are bounded knapsack problems (BKPs). This means that each element can be used only as often as needed. This leads to the assumption that

$$b_i \leq \left\lfloor \frac{c}{w_i} \right\rfloor, \qquad j = 1, \ldots, n \tag{3.14}$$

as any excessive demand can immediately be reduced. The BKP is an NP hard problem, but there exist efficient methods to solve it optimally. A popular approach is transforming the BKP to a binary knapsack problem (KP), and then solving the KP, see 2.0.6. A branch & bound algorithm is described in Section 2.0.8, while an approximation algorithm is investigated in Section 2.0.7. A dynamic programming algorithm is be presented in Section 3.3.6, which is the basis for algorithm BKP-Generator (Section 3.3.7).

3.3.6 Dynamic Programming

In publications, there is one more advanced dynamic programming algorithm which heavily makes use of multiple elements of the same type [Pfe99], referred to as Improved-DP. In principle, the idea is similar to the transformation to a KP, since each copy of an element is added separately to an existing partial solution. However, all copies of the same element type are added together in an efficient way. To be more precise, after processing element types $1, \ldots, j-1$ we try to add a new element of type j to every solution attached to an entry of the dynamic programming solution. If $z_j(d + w_j) = z_{j-1}(d) + p_j$, i.e. adding element j to the previous solution improves that solution, we immediately try to add another copy of the same element type to $z_{j-1}(d)$, therefore attempting to improve $z_{j-1}(d + 2w_j)$. In this way, we chain elements of the same type together for as long as there are elements (at most b_j), there is capacity or of course there is an update. If a sequence of elements is terminated because all b_j copies of element j were successfully added, we must consider the possibility that a chain might be added at a later point than d, such that a capacity value higher than $d + b_j w_j$ can possibly be improved. The most promising starting point for a new chain of elements j then lies somewhere between $d + w_j$ and $d + b_j w_j$, and can be computed by keeping track of the previous update sequence. Simplified pseudo code for the Improved-DP is shown below in Algorithm 3.8, adapted from [KPP04].

Algorithm 3.8: Improved-DP

```
1  z_j(d) := 0 for d = 0,...,C and j = 0,...,n;
2  sort elements in ascending order by efficiency (p_j/w_j);
3  foreach item j do
4      initialize auxiliary array;
5      for r := 0,...,w_j − 1 do // all residual values of C/w_j
6          d_0 := r;
7          d := r + w_j;
8          l := 1;
9          while d ≤ C do
10             if z_{j−1}(d) < z_{j−1}(d_0) + lp_j then
11                 z_j(d) := z_{j−1}(d_0) + lp_j;
12                 insert d in auxiliary list;
13                 d := d + w_j;
14                 if l < b_j then
15                     l := l + 1;
16                 else
17                     set d_0 as best candidate from auxiliary list;
18                     l := (d − d_0)/w_j;
19                 end
20             else
21                 z_j(d) := z_{j−1}(d);
22                 d_0 := d;
23                 l := 1;
24                 d := d + w_j;
25                 delete auxiliary list
26             end
27         end
28     end
29 end
```

The variable d_0 is used as the starting capacity for a sequence, and is only changed in a current run in line 17 if all elements of type j are used, i.e. $l = b_j$. d is used as the capacity for a considered update operation, and the variable l is the number of copies of element j are added to the capacity d_0. The most relevant line to note is line 5. For each element j we iterate over all residual values c/w_j, and at each of these values we chain the current item. Line 10 checks for an improvement, and then line 11 does the update. Since the process of keeping the auxiliary list up to date does not change the fundamental logic of the algorithm, the details of lines 4 and 17 are omitted.

Complexity

The run time depends both on the capacity C and the number of elements n. Firstly the algorithm runs over all elements in the for loop in line 3, giving a time complexity of $O(n)$. The second relevant loop is in line 5, where we run over the remainders r of each element type j. Each of the w_j executions has a run time of $O(C/w_j)$, given by the while loop in line 9. That means line 5 has a time complexity of $O(C)$, resulting in a total worst case run time of $O(Cn)$. The space requirement for storing the solution vector is $O(Cn)$, as a vector $z_j(d)$ for each element type must be stored.

3.3.7 BKP-Generator

Algorithm 3.8 has one issue concerning the implementation, and that is that it is not particularly suited to using discretization points, as used in algorithm 3.2. Although discretization points could be used to reduce the number of residual values, we would have to take the residual values of *all* discretization points for each item, thereby introducing new discretization points altogether. This is because otherwise linear combinations of 2 or more elements would be ignored. Moreover, as a consequence of each newly introduced discretization point in the residual values, potentially new discretization points follow. For a simple example, let $n = 2$ with $w_1 = 3$, $w_2 = 7$ and $C = 10$. The initial set of discretization points is then $\mathcal{P} = \{0, 3, 6, 7, 9, 10\}$. The residual values for $i = 1$ would then have to be $r = \{0, 1\}$ for the chain to reach the discretization point $p_4 = 7$. Already we see that the chain in this case passes point $p = 4$, which, like point $p = 1$ is not one of the discretization points. Unfortunately, this natural incompatibility just gets worse the larger an instance is, and the more relevant optimizations become. Since our instances and element types have very large capacities and weights, the use of discretization points is paramount to reduce computation time. Therefore we introduce a new dynamic programming algorithm, based on the idea in Improve-DP to work through the solution vector item-wise, which is more suited to work with discretization points. The basic idea is to fill the knapsack with each item, always checking the discretization points for a possible improvement. It is done item-wise so a separate vector, $l(d)$ can be kept to update the uses of the element at a specific discretization point (capacity). This essentially means that a sequence of element j is either continued at each discretization point, or started. The pseudo code for algorithm 3.9 is below.

The algorithm begins by sorting the elements by decreasing efficiency p_i/w_i. This means that the algorithm starts by packing the most efficient element first. Initially, the amount of uses of the current element at each discretization point is set to 0, over the loop on line 8 and 9. Line 11 and 12 are standard dynamic programming, in which for each capacity we check if the current element produces an improvement. Line 13 and 22 represent the BKP, since a distinction is made between adding an element or not, based on demand. If another element can be added to the solution, its usage is increased by 1 for that capacity (line 15), and the solution value is also updated (line 14). If the demand has been reached however, we either update without adding an item, or we don't update at all because the old value is better without adding a new item.

Algorithm 3.9: BKP-Generator

```
1  z(d) := 0 for d = 0, ..., C;
2  pointToSolution = ∅;
3  bestCapacity = 0;
4  bestValue = 0;
5  sort elements in descending order by efficiency (p_j/w_j);
6  getDiscretizationPoints;
7  foreach item j do
8      foreach capacity d in discretizationPoints do
9          l(d) := 0;
10     end
11     foreach capacity d in discretizationPoints do
12         if z(d) < z(d − w_j) + p_j then
13             if l(d − w_j) < b_j then
14                 z(d) := z(d − w_j) + p_j;
15                 l(d) := l(d − w_j) + 1;
16                 pointToSolution(d) = pointToSolution(d − w_j) ∪ {j};
17                 if z(d) > bestValue then
18                     bestValue = z(d);
19                     bestCapacity = d;
20                 end
21             else
22                 if z(d) < z(d − w_j) then
23                     z(d) := z(d − w_j);
24                     l(d) := l(d − w_j);
25                     pointToSolution(d) = pointToSolution(d − w_j);
26                 end
27             end
28         end
29     end
30 end
```

Let $n = 2$ with $w_1 > w_2$ and $p_1/w_1 > p_2/w_2$. Figure 3.2 shows a possible updating scenario of algorithm 3.9. After all computations for element $i = 1$ are done, here called e_1, one possible solution is shown in figure 3.2a. In the next step of the algorithm, element e_2 is placed in the first possible discretization point where it increases the solution value, as shown in figure 3.2b. However, the algorithm has no ability to retroactively increase the value of each successive placement of e_1, which were calculated in the previous iteration. This problem is solved by the fact that using the equivalent solution in figure 3.2d, the algorithm can update the solution vector correctly. Indeed, the solution seen in 3.2b is still calculated iteratively, using the parts from figure 3.2c. This is due to the fact

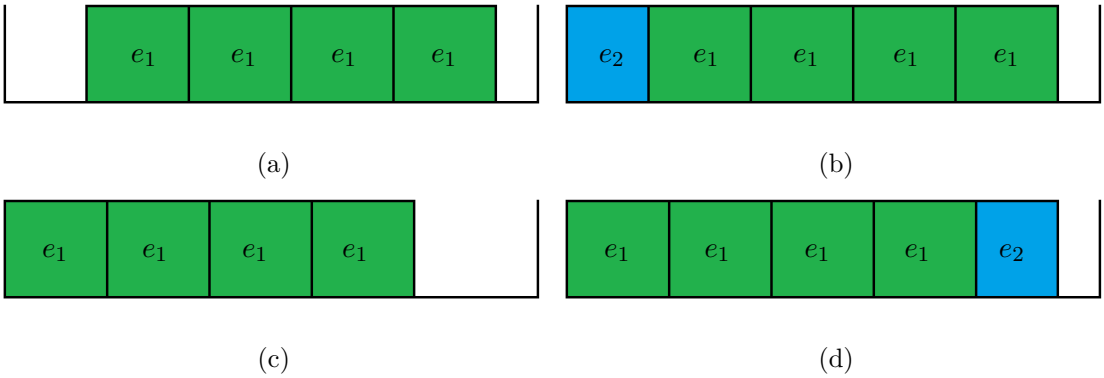


Figure 3.2: Equivalent solutions for the BKP.

that all solutions have an equivalent one which is "flushed left", i.e. compact such that all of the unused space is at the end of the knapsack. Sorting the elements by efficiency is necessary so that after one element is finished computing, the next element can only improve on the previous compacted solutions if it uses up some previously unused space. For retrieving the solution, it is required to store the solution for each discretization point, and store the currently best discretization point.

Complexity

The run time depends both on the capacity C and the number of elements n. As discussed previously, the time complexity of algorithm 3.5, getDiscretizationPoints, is $O(Cn)$. Line 7 has a complexity of $O(n)$, and in the case that all points are discretization points, the inner loop has a worst case run time of $O(C)$, resulting in $O(Cn)$ run time. Similarly to EDUK, the use of reduced discretization points increases the final worst case run time to $O(Cn\log(C))$. As a pricing problem, this algorithm will be run for every available strip height, i.e. discretization point in the sheet height, giving the overall pricing problem per execution a run time of $O(WHn\log(W))$, where H is the sheet height, and W is the sheet width. Since the solution vector is two dimensional, the space requirement is $O(Cn)$, although the solution can also be reconstructed by backtracking a one dimensional solution vector, requiring $O(Cn)$ additional time and $O(C)$ space instead.

3.4 The 3-Staged 2CS

Whereas in the case of $K = 2$ the elements are separated by their respective heights, in the case of $K = 3$ a strip has access to all elements. The pricing problem is now a special variant of a 2-staged two-dimensional knapsack problem. The literature in this case focuses more on the SLOPP producing entire sheets, not on the strip characteristic of the problem, i.e. the linear cost coefficient of the height variable. As such, we present our own algorithms to solve the 2-staged strip generating pricing problem. One algorithm

is a dynamic programming approach with a focus on returning the best possible strip (section 3.4.1), while two heuristic variants are presented in section 3.4.2.

3.4.1 Dynamic Programming Generator

Our dynamic programming implementation, called DP-Generator, borrows heavily from algorithm 3.9. It is essentially the same algorithm, simply upgraded for two stages. Whereas BKP-Generator solves one knapsack per available height, DP-Generator is separated into two stages. In the first, a knapsack is solved for every available *width*, i.e. after all elements are separated by width, the corresponding knapsack problem is solved. This results in an optimal solution for every element width and strip height, called *stacks*. In the second stage, these results are used to calculate the optimal solution for every strip height. The parallels between BKP-Generator and DP-Genertaror in this stage are that while algorithm 3.9 uses a set of elements with equal height to the strip height h_p, DP-Generator uses the best stack to fit into the given strip height, i.e. $h_s \leq h_p$ where h_s is the stack height. The pseudo code can be seen in algorithm 3.10.

The algorithm first calculates the discretization points for the sheet width, as seen in algorithm 3.5. During that process, it also groups elements by width, called $widthToElements$. This is a process that only has to be done once, meaning following calls to DP-Generator do not need to recalculate the discretization points for the width. The next step is to generate the relevant stacks in algorithm 3.16, seen at the end of this chapter, sorted by profit. This can be seen as solving BKP-Generator once for every width. Algorithm getDynProgStacks (3.16) also returns the relevant strip heights, i.e. the discretization points of the sheet height, one for each stack. After generating the stacks, the algorithm iterates over all strip heights. The algorithm 3.11 is called to choose the best stack for every width, and sorts them by efficiency p_j/w_j. The remaining algorithm is the same as algorithm 3.9. One point to note however, is the meaning of b_j in the context of the 2-staged knapsack problem. While in BKP-Generator it was clearly the demand of element j, here it is the "demand" of a stack j. The demand of stack j is defined here as the lowest demand to cost ratio for all elements in stack j (rounded down to the nearest integer), where the cost of element i is the number of times element i appears in stack j. This is possible because for every width, only one stack is chosen per strip, so that keeping track of individual element demands is unnecessary.

Complexity

The run time complexity of algorithm 3.10 depends on the sheet height H, sheet width W, and number of elements n. Algorithm 3.16 needs to solve a knapsack with capacity H in the worst case for each width in W. The time complexity of solving a bounded knapsack was already established as $O(Cn)$, so algorithm 3.16 has a worst case time complexity of $O(WHn)$. The remainder of algorithm 3.10 also solves a bounded knapsack with a capacity of W for each height H. This results in the same run time of $O(WHn)$, although in practice it is much closer to this than getDynProgStacks.

Algorithm 3.10: DP-Generator

```
1  getDiscretizationPoints() ;  // for stacks and width, only once
2  separateItems() ;                                  // only done once
3  getDynProgStacks();
4  foreach int currentHeight in heightDPs do
5  |  z(d) := 0 for d = 0, ..., c;
6  |  getCandidateStacks(currentHeight);
7  |  foreach stack j in candidateStacks do
8  |  |  foreach capacity d in discretizationPoints do
9  |  |  |  l(d) := 0;
10 |  |  end
11 |  |  foreach capacity d in discretizationPoints do
12 |  |  |  if z(d) < z(d − w_j) + p_j then
13 |  |  |  |  if l(d − w_j) < b_j then
14 |  |  |  |  |  z(d) := z(d − w_j) + p_j;
15 |  |  |  |  |  l(d) := l(d − w_j) + 1;
16 |  |  |  |  else
17 |  |  |  |  |  if z(d) < z(d − w_j) then
18 |  |  |  |  |  |  z(d) := z(d − w_j);
19 |  |  |  |  |  |  l(d) := l(d − w_j);
20 |  |  |  |  |  end
21 |  |  |  |  end
22 |  |  |  end
23 |  |  end
24 |  end
25 end
```

Algorithm 3.11: getCandidateStacks

```
1  candidateStacks = ∅;
2  foreach width w in widthToStacks do
3  |  foreach stack s in w do
4  |  |  if h_s < currentHeight then
5  |  |  |  add s to candidateStacks;
6  |  |  |  break;
7  |  |  end
8  |  end
9  end
10 sort candidateStacks by efficiency (p_i/w_i);
```

3.4.2 Heuristic-Generator

Most literature on sheet based heuristics considers the height of the sheet fixed. That means an adaption to strip generation either adds another heuristic aspect, the strip height, or we need to solve the heuristic for every available height. Given these considerations, we present our own heuristic variants of our dynamic programming implementation. Note that both heuristics call DP-Generator after sufficiently many failed attempts at finding a strip with negative reduced cost.

The first heuristic uses algorithm 3.10, but randomizes the strip height. As a variable, the number of random strip heights is given, where each strip height is taken from a distinct set of heights. Our implementation of the first variant of the Heuristic-Generator can be seen in algorithm 3.12.

Algorithm Heuristic-Generator takes advantage of the structure of algorithm 3.10 to randomize the heights. The strategy followed here is of separating the heights into sectors, and choosing one height per sector. This minimizes the randomness in direct correlation with an increase of variable *heuristic*. Lines 5 to 10 randomize the available heights. Essentially, the set of discretization points is separated into "*heuristic*" many subsets of equal size, each of which then give one height candidate. This is done to avoid occurrences where all random heights offer particularly bad results. Furthermore, a weighted function is kept to ensure an overall even selection of heights, i.e. to distribute the random heights more evenly over all strip heights. Also note the use of a different stack generating algorithm in line 3, described in algorithm 3.13. Instead of using dynamic programming to generate just the best stacks, this algorithm generates all possible stacks, meaning all combinations of items, and therefore more possible strip heights. This opens the heuristic up to solutions otherwise not available to algorithm 3.10. Moreover, this means the stacks must only be generated once, and every subsequent call of Heuristic-Generator simply needs to update the profit values of each stack.

Although algorithm getStacks can fall victim to a combinatorial explosion, in practice this is unlikely. Especially in cases where we hope to achieve good results, i.e. instances of few different element types, the combinatorial explosion is of little concern. Given these considerations, it is not surprising that benchmarks show that updating a larger list of stacks performs better than recalculating the stacks, as is done in algorithm 3.10. However, recalculating has the added benefit of a much smaller list of heights.

Heuristic-Generator 2

After randomizing the height, another aspect that can be taken into consideration are the profit values for the elements (the duals). The idea behind the second heuristic is to attempt to predict how the dual variables change, based on the selected strip in the previous iteration. This basically means reducing the profit of the used stacks directly, while leaving the other stacks unaffected, mostly because the real dual values vary "arbitrarily", due to the dependence on the resolved master problem. Because the effect is similar to the previous heuristic, in that the resulting algorithm will distribute

Algorithm 3.12: Heuristic-Generator

```
   Input: how many heights should be calculated; heuristic
 1 getDiscretizationPoints() ;        // width and height, only once
 2 separateItems() ;                                  // only done once
 3 getStacks();                                       // only done once
 4 randomheights = ∅;
 5 foreach int i in heuristic do
 6 |   min = (sheetHeight/heuristic) * i + 1;
 7 |   max = (sheetHeight/heuristic) * (i + 1);
 8 |   height = random(min, max);
 9 |   randomHeights = randomHeights ∪ {height};
10 end
11 foreach int currentHeight in randomHeights do
12 |   z(d) := 0 for d = 0, ..., c;
13 |   getCandidateStacks(currentHeight);
14 |   foreach item j in candidateStacks do
15 |   |   foreach capacity d in discretizationPoints do
16 |   |   |   l(d) := 0;
17 |   |   end
18 |   |   foreach capacity d in discretizationPoints do
19 |   |   |   if z(d) < z(d − w_j) + p_j then
20 |   |   |   |   if l(d − w_j) < b_j then
21 |   |   |   |   |   z(d) := z(d − w_j) + p_j;
22 |   |   |   |   |   l(d) := l(d − w_j) + 1;
23 |   |   |   |   else
24 |   |   |   |   |   if z(d) < z(d − w_j) then
25 |   |   |   |   |   |   z(d) := z(d − w_j);
26 |   |   |   |   |   |   l(d) := l(d − w_j);
27 |   |   |   |   |   end
28 |   |   |   |   end
29 |   |   |   end
30 |   |   end
31 |   end
32 end
```

its selection of strips over different heights evenly, the thinking can be "inverted". Instead of promoting unused stacks by lowering the profit of used stacks, an iteration of real dual variables can produce multiple random height strips. This means leaving the profit values unchanged for more than one strip. The result is two contrasting strategies: Heuristic-Generator produces one hopefully good strip per set of duals, while Heuristic-Generator 2 produces multiple random strips per set of duals. In other words, a higher number of

Algorithm 3.13: getStacks

```
1  discretizationPoints = ∅;
2  foreach vector ⟨item⟩ width in widthToElements do
3      changed = true;
4      oldStacks = 0;
5      while changed do
6          changed = false;
7          newStacks = ∅;
8          foreach item i in width do
9              foreach stack s in oldStacks do
10                 if h_s + h_i < H then
11                     if newStacks not contains s ∪{i} then
12                         add s ∪ {i} to newStacks;
13                         add (h_s + h_i) to heightPoints;
14                         changed = true;
15                     end
16                 end
17             end
18         end
19         add newStacks to stacks;
20         oldStacks = newStacks;
21     end
22     sort stacks by profit p_i;
23     add stacks to widthToStacks;
24 end
```

lower quality strips are produced, offloading computational complexity from the pricing problem onto the master problem.
Due to the algorithmic similarities between the two, pseudo code for Heuristic-Generator 2 is not explicitly shown. Instead, in algorithm 3.14, the update function is shown.

Algorithm updateDuals essentially does two checks. If the previous strip did not lead to an improvement in the master problem, the dual variables and stacks are updated. Otherwise, if the average absolute deviation from each dual value is larger than the given *heuristic* value, the stacks are also updated.

3.5 Integrality Heuristic

Since the model described in section 3.2 only solves the LP relaxed problem, the resulting solution contains possibly fractional values for not only variable y_j, but also for the values of x_{pj}. Algorithm 3.1 therefore calls an integrality heuristic to make the fractional solution integral. This heuristic is in place of a more elaborate branch & price algorithm,

Algorithm 3.14: updateDuals

```
   Input: current objective value, new dual values, variable heuristic
 1 if prevObj = curObj then
 2 |  updateStacks;
 3 end
 4 calDeviations() ;                          // calculate avg deviation
 5 if deviation > heuristic then
 6 |  updateStacks;
 7 end
 8 prevObj = curObj
```

for which we expect better results, albeit at greater computational cost. Cintra et al. [CMWX08] propose a heuristic for their sheet based algorithm, on which our heuristic is based. Pseudo code for our integrality heuristic can be seen in algorithm 3.15.

The single largest difference between our heuristic and that of Cintra et al. is represented in line 8. After every variable that is fixated (line 6), the model is resolved, whereas Cintra et al. [CMWX08] round down all variables, and solve for the induced remaining problem. Solving after every fixation does not cost much computation time, thanks to the "warm start" attribute of simplex. This leads to dramatically better results, due to the fact that the solution remains feasible throughout. The infeasibility is caused in part by the symmetry breaking constraint, or any other ordering, which break when some sheets can be arbitrarily reduced far more than others. Moreover, it is obvious that less restraining leads to a better result. Setting $y_j = 1$ in line 7 helps the algorithm fill partially filled sheets first. If we know a strip will be in the sheet, optimally the entire sheet should be used. Without this distinction, starting a new sheet or remaining in the same sheet would contribute equally to the objective function, i.e. setting $y_j = 1$ creates unused, "free" space for the column generation to use. The final point to note is that in line 5 we select only values greater or equal to 1 to round down to. The logic behind this decision is that disallowing a strip just prompts the pricing problem to either generate the exact strip in another sheet, or generate an almost identical one in the same sheet. It further promotes infeasible ILP models, since the strips that are used less than once are typically also the most necessary to make the solution feasible. This leads to solutions of algorithm 3.15 that are not yet integral, in the sense that there are patterns with $0 < x_{pj} < 1$. In these cases, the insertion heuristic from section 3.1 is used to insert the remaining elements into the almost integral solution. Further variations are investigated in chapter 5.

Complexity

The run time complexity of algorithm 3.15 depends on the upper bound of the number of sheets n, and the number of strips assigned per sheet $|P|$, resulting in $O(n|P|)$ time. The worst case of the number of strips per sheet can be exponential in the number of

Algorithm 3.15: integralityHeuristic

Input: Fractional 2CS solution *model*, pricing problem *pricing*, variable *tries*
Output: Integral 2CS solution *model*

```
1  attempts = 0;
2  while attemps++ < tries do
3      for j = 1 to n do
4          foreach p ∈ P do
5              if x_pj ≥ 1 then
6                  x_pj = ⌊x_pj⌋ ;                    // fixate value
7                  y_j = 1;                           // fixate value
8                  model.solve();
9              end
10         end
11     end
12     if !fractional then break;
13     duals = model.getDuals();
14     while pricing.solve(duals) do
15         newColumn = pricing.getSolution();
16         model.add(newColumn);
17         model.solve();
18         duals = model.getDual();
19     end
20 end
21 solution = insertionHeuristic();          // insert remaining items
22 return solution;
```

elements there are, however.

Algorithm 3.16: getDynProgStacks

Input: map of widths to vector of elements with equal widths (widthToElements), map of widths to discretization points (widthToDiscretizationPoints)

Output: map of widths to vector of stacks sorted by profit, discretization points of sheet height

```
1  widhtToStacks = ∅;
2  foreach vector ⟨item⟩ width in widthToElements do
3      discretizationPoints = widthToDiscretizationPoints(width);
4      widthToStacks(width) = ∅;
5      z(d) := 0 for d = 0, ..., C;
6      foreach item j in width do
7          pointToStack = ∅;
8          foreach capacity d in discretizationPoints do
9              l(d) := 0;
10             pointToStack(d) = ∅;
11         end
12         foreach capacity d in discretizationPoints do
13             if z(d) < z(d − w_j) + p_j then
14                 if l(d − w_j) < b_j then
15                     z(d) := z(d − w_j) + p_j;
16                     l(d) := l(d − w_j) + 1;
17                     pointToStack(d) = pointToStack(d − w_j) ∪ {j};
18                 else
19                     if z(d) < z(d − w_j) then
20                         z(d) := z(d − w_j);
21                         l(d) := l(d − w_j);
22                         pointToStack(d) = pointToStack(d − w_j);
23                     end
24                 end
25             end
26         end
27     end
28     currentBest = 0;
29     foreach capacity d in discretizationPoints do
30         if z(d) > currentBest then
31             widthToStacks = widthToStacks ∪ pointToStack(d);
32             currentBest = z(d);
33             heightDPs = heightDPs ∪ {d};
34         end
35     end
36 end
37 sort heightDPs;
38 return widthToStacks;
```

CHAPTER 4

Implementation

The SSCG is implemented in C++ and compiled using gcc 4.8.4. It can also be compiled using Visual Studio 2013 or 2012. The ILP was modeled and solved using IBM Cplex 12.6. It is built on a framework by Frederico Dusberger [DR15b] [DR15a], which provides the insertion heuristic used for the initial set of columns, as well as all input/output management and data handling.
The program can be called using a number of input values. An example program call would look like the following:

```
k2csv pfile /home1/e0826666/inst/221.cut
alg 1 K 3 heur 0 symmetry 1 heur2 0 runs 3
allow_elem_rotation 0 allow_sheet_rotation 0
```

In this case, "pfile" refers to the instance file, "alg" refers to the algorithm to be used, "K" refers to the number of stages, while "heur" is the number of strips Heuristic-Generator should generate, 0 meaning not to use it at all. The "alg" input value refers to the algorithm of the framework, in which 1 refers to the SSCG. Finally, "symmetry" refers to whether or not to use the symmetry constraints, "heur2" refers to using Heuristic-Generator 2 or not, and "runs" is the number of times the integrality heuristic tries to add new columns after finishing rounding all values. More specifically, if runs= 3, the combination of adding columns and running the integrality heuristic on the result repeats 3 times. Given the way the generators are implemented, all input variables can be used at the same time. This means technically, Heuristic-Implemenation 2 can be run while generating the best strip out of 5 (heuristic = 5), with or without the symmetry constraints. The final two input values are necessary, since the framework supports rotated elements, SSCG however does not. This is relevant in detecting feasible instance files, and a consequence of the initial feasible solution.

Concerning the structure of the program, it follows the structure of the framework. The class consisting of the SSCG is called *ColumnGeneration* and inherits from the *Algorithm* class. All pricing problems inherit from the new class *PricingProblem*, which holds most relevant data for the generators. This includes the constructor, an *update* function which is used to update the duals, and the *solve* function, which returns true if a strip with negative reduced cost is found, and false otherwise. Finally, if a solution is found, it is returned using the *getSolution* function of the *PricingProblem* class. It is also of note that like in Section 3.2.1, the pricing problem is implemented as a *maximization* problem, meaning the signs of the dual variables are reversed.

CHAPTER 5

Results

All experimental tests were run on the *Sun Grid Engine* (SGE), version 6u2, on the computation grid of the Algorithms and Complexity Group at the Technical University of Vienna. It uses a Quad Core Intel Xeon E5649, with 2.53 GHz.
A popular benchmark for bin packing problems in literature is presented below. There are 10 classes, each of which having 5 subclasses with $n \in \{20, 40, 60, 80, 100\}$. The benchmark therefore consists of 50 unique random instance types.
The first four classes were proposed by Martello and Vigo [MV98], and are based on generating four different element types:

- Type 1: w_j uniformly random in $[\frac{2}{3}W, W]$, h_j uniformly random in $[1, \frac{1}{2}H]$.
- Type 2: w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[\frac{2}{3}H, H]$.
- Type 3: w_j uniformly random in $[\frac{1}{2}W, W]$, h_j uniformly random in $[\frac{1}{2}H, H]$.
- Type 4: w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[1, \frac{1}{2}H]$.

Each class k ($k \in \{1, 2, 3, 4\}$) is then obtained by generating an element of type k with a probability of 70%, while the remaining 3 element types have a probability of 10% each. In these classes, the sheet size is $W = H = 100$.
The following six classes were proposed by Berkey and Wang [BW87]:

- Class 5: $W = H = 10$, with w_j and h_j uniformly random in $[1, 10]$.
- Class 6: $W = H = 30$, with w_j and h_j uniformly random in $[1, 10]$.
- Class 7: $W = H = 40$, with w_j and h_j uniformly random in $[1, 35]$.
- Class 8: $W = H = 100$, with w_j and h_j uniformly random in $[1, 35]$.

		BKP-Gen					EDUK-Gen			
Class	n	t	Cls	LP	Sol	$Heur$	t	Cls	LP	Sol
	4	48.7	0.8	1060.7	1262.0	**1259.0**	46.7	0.8	1060.7	1262.0
	8	166.8	51.0	1949.2	2340.7	**2328.3**	179.9	51.0	1949.2	2340.7
1	12	495.8	114.8	2786.0	3158.7	**3143.1**	428.6	114.8	2786.0	3158.7
	16	1093.4	368.6	4075.6	4866.5	**4857.1**	887.6	368.6	4075.6	4866.5
	20	972.8	290.2	4395.7	5225.2	**5187.5**	802.8	290.2	4395.7	5225.2
	4	61.6	0.1	1085.3	1181.7	**1165.4**	53.7	0.1	1085.3	1181.7
	8	217	0.1	1995.4	2134.5	**2097.4**	222.7	0.1	1995.4	2134.5
2	12	537.3	0.5	3074.7	3261.6	**3222.9**	561.9	0.5	3074.7	3261.6
	16	1049.9	1.1	4107.5	4329.2	**4260.8**	1112.1	1.1	4107.5	4329.2
	20	1336.2	0.6	4610.0	4875.8	**4714.2**	1154.0	0.6	4610.0	4875.8
	4	343.1	0.3	2322.3	2885.7	**2851.6**	345.2	0.3	2322.3	2885.7
	8	942.1	0.8	4416.4	5816.0	**5799.0**	1061.5	0.8	4416.4	5816.0
3	12	NaN	NaN	NaN	NaN	NaN	1723.2	0	6587.5	7916.0
	16	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	20	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	4	24.4	0.6	770.3	855.3	**849.5**	28.7	0.6	770.3	855.3
	8	188.9	15.3	1683.4	2036.5	**1969.8**	195.9	15.3	1683.4	2036.5
4	12	271.1	41.1	2255.9	2578.4	**2536.4**	278.0	41.1	2255.9	2578.4
	16	543.4	64.8	2987.5	3366.6	**3226.0**	558.9	64.8	2987.5	3366.6
	20	806.1	79.9	3641.7	4132.2	**3974.6**	764.9	79.9	3641.7	4132.2

Table 5.1: Average results for BKP-Generator and EDUK-Generator for Classes 1 through 4, for each $n \in \{20, 40, 60, 80, 100\}$. NaN refers to timeouts or out of memory exceptions in all 10 instances.

- Class 9: $W = H = 100$, with w_j and h_j uniformly random in $[1, 100]$.
- Class 10: $W = H = 300$, with w_j and h_j uniformly random in $[1, 100]$.

For each class and value of n, ten instances are generated, resulting in 500 instances overall. We adapted these instances to better fit our demand of few types, but many items. Firstly, for all instances, the sheet and element sizes were multiplied by 100, i.e. $H_{new} = 100H$ and $W_{new} = 100W$. Also, the number of element types was reduced, such that for each $n \in \{20, 40, 60, 80, 100\}$ we have $n_{new} = \frac{n}{5}$. Finally, a constant demand $D = 1000$ was added to each item.

In Tables 5.1 and 5.2, t is the average time measured in seconds, Cls is the average number of columns added, LP is the average optimal LP solution after adding the columns, while Sol is the average final solution. $Heur$ refers to the average solution of the insertion heuristic.
Looking at Tables 5.1 and 5.2, we can see that both algorithms have identical results. This is not surprising given the extremely high demand of the items. There exists no case

		BKP-Gen					EDUK-Gen			
Class	n	t	Cls	LP	Sol	$Heur$	t	Cls	LP	Sol
	4	82.3	6.9	1493.0	1665.9	**1654.5**	85.7	6.9	1493	1665.9
	8	419.6	33.0	2509.7	2776.1	**2705.6**	476.3	33.0	2509.7	2776.1
5	12	1009.9	72.3	3916.4	4268.1	**4217.6**	1091	72.3	3916.4	4268.1
	16	1274.6	130.8	4535.3	4976.2	**4831.2**	1364.8	130.3	4535.3	4976.2
	20	1381.3	3.5	4491.7	4889.5	**4564.5**	1489.6	3.5	4491.7	4889.5
	4	0.1	25.2	144.5	149.9	**148.0**	0.2	25.2	144.5	149.9
	8	0.5	39.3	239.3	250.8	**245.8**	0.5	39.3	239.3	250.8
6	12	1.7	77.0	437.3	458.9	**447.2**	1.6	77.0	437.3	458.9
	16	2.4	79.1	519.2	543.5	**530.4**	2.5	79.1	519.2	543.5
	20	8.6	165.3	677.8	709.3	**688.0**	10.2	165.3	677.8	709.3
	4	45.7	2.0	1039.9	1223.4	**1208.1**	45.9	2.0	1039.9	1223.4
	8	257.7	81.6	1760.8	2110.9	**2060.9**	250.2	81.6	1760.8	2110.9
7	12	717.7	12.2	3323.2	3934.6	**3817.6**	711.4	12.2	3323.2	3934.6
	16	734.4	31.0	3480.9	4071.1	**3900.9**	695.1	31	3480.9	4071.1
	20	1043.2	34.0	4015.5	4480.0	**4303.0**	933.1	34.0	4015.5	4480.0
	4	0.2	0.3	150.3	162.8	**159.0**	0.2	0.3	150.3	162.8
	8	0.7	18.1	242.3	257.0	**252.4**	0.7	18.1	242.3	257.0
8	12	2.2	42.3	460.6	488.0	**474.1**	2.1	42.3	460.6	488.0
	16	3.2	76.4	541.4	579.4	**559.7**	3.1	76.4	541.4	579.4
	20	10.2	154.7	695.2	734.2	**713.4**	9.8	154.4	695.2	734.2
	4	86.5	3.0	1347.3	1606.2	**1588.6**	78.7	3.0	1347.3	1606.2
	8	272.3	6.0	2215.9	2616.2	**2548.8**	255.2	6.0	2215.9	2616.2
9	12	754.9	8.6	3856.6	4400.1	**4273.8**	670.1	8.6	3856.6	4400.1
	16	1065.5	37.5	4300.8	5021.5	**4763.5**	1040.4	37.5	4300.8	5021.5
	20	1277.2	1.7	4475.8	4815.0	**4710.3**	1265	1.7	4475.8	4815.0
	4	0.1	0.9	126.2	133.8	**132.2**	0.1	0.9	126.2	133.8
	8	0.4	4.7	200.7	213.3	**209.9**	0.4	4.7	200.7	213.3
10	12	1.3	7.6	393.0	412.4	**403.7**	1.2	7.6	393.0	412.4
	16	2.1	39.3	465.6	498.0	**486.1**	2.0	39.3	465.6	498.0
	20	3.9	50.1	609.4	643.7	**625.4**	3.8	50.1	609.4	643.7

Table 5.2: Average results for BKP-Generator and EDUK-Generator for Classes 5 through 10, for each $n \in \{20, 40, 60, 80, 100\}$.

where EDUK would find an actually infeasible strip. Also to note is the fact that the best results, i.e. the ones closest to the solution from the Insertion Heuristic, are those with the fewest amounts of added columns. This is due to the integrality heuristic going back to the original Insertion Heuristic solution, if the master problem does not change too much. The Insertion Heuristic is never beaten, which is also not surprising. The pricing problem cannot produce very many columns worth adding, given the stage restriction. The LP relaxation seems good, however. We can also see that class 3 is practically not solvable. Upon closer inspection, most elements require an entire sheet, blowing up the ILP model. Overall, classes 1 through 4 all end up putting more computational burden on the master problem than the pricing problem in the 2 staged case, due to the relatively large element size.

3 Stages

Given the results for $K = 2$, the benchmarks do not seem promising for the case of $K = 3$. When $K = 2$ we already get a multitude of unfinished instances due to the memory consumption of Cplex. This is evident not just in class 3 ("NaN" refers to instance that never finished), but in a number of other classes as well. The problem gets exacerbated when the pricing problem starts adding many columns to the model, as is expected in the 3 staged problem. Indeed, first tests did not return good results, with a great deal not finishing. As such, we chose another set of popular instances from literature. Alvarez-Valdes et al. [AVPT02] uses a selection of instances randomly generated, or taken from other literature. Although they are used not for the 2CS, but as an optimization packing problem, i.e. every element has a value v_i, over which a maximum is tried to be packed. H is given by Herz [Her72], HZ1 is given by Hifi and Zissimopoulos [HZ96], $M1$-$M5$ appears in Morabito et al. [MAA92]. $UU1$-$UU10$ are randomly generated in Fayard et al. [FHZ98] as follows. The dimensions of the sheets are random in $[500, 4000]$, the dimensions of the pieces are in intervals $[0.01H, 0.7W]$, and the number of element types is random in $[25, 60]$. To make the instances a 2CS problem, we added a demand of 50 to each element type.
Further instances, these constrained (i.e. the element types have an upper bound, which we use as demand), include $OF1$ and $OF2$ from Oliveira and Ferreira [OF90] and W from Wang [Wan83]. Instances $CU1$-$CU11$ are generated as follows. The dimensions of the sheet are random in $[100, 1000]$, the dimensions of the pieces are in the intervals $[0.01H, 0.7W]$, and the number of element types is again between $[25, 60]$. The demand b_i for element i is $\max\{1, \min\{10, random(\gamma)\}\}$, where $\gamma = \lfloor H/h_i \rfloor \lfloor W/w_i \rfloor$.
$ATP30$-$ATP39$ are again taken directly from Alvarez-Valdes et al, which are randomly generated large scale test problems. The number of element types is in the interval $[30, 60]$, the sheet size is in $[1500, 3000]$, and the dimensions of the piece w_i and h_i in the intervals of $[0.05W, 0.4W]$ and $[0.05H, 0.4H]$ respectively.
The remaining instances are from Cung et al. [CHC00]. The CHL and $Hchl$ instances are both randomly generated. For these, the dimensions are as follows. element dimensions h_i and w_i are $[0.1H, 0.75H]$ and $[0.1W, 0.75W]$, respectively. The demands are such that $b_i = \min\{\rho_1, \rho_2\}$, where $\rho_1 = \lfloor H/h_i \rfloor \lfloor W/w_i \rfloor$ and ρ_2 is in the interval $[1, 10]$.

	Symmetry = 0					Symmetry = 1				
Instance	t	Cls	LP	LP_{CG}	Sol	t	Cls	LP	LP_{CG}	Sol
2s	0.04	11	2.29	1.70	3	0.03	11	2.29	1.70	3
3s	0.12	2	20.94	20.86	28	0.12	7	20.94	20.86	**26**
A1s	0.30	34	19.78	17.98	25	0.29	35	19.78	17.98	25
A2s	0.35	40	11.47	10.47	13	0.45	50	11.47	10.47	13
A3	0.50	45	7.79	6.82	8	0.40	37	7.79	6.84	8
A4	1.18	59	4.34	4.03	5	1.23	60	4.34	4.03	5
A5	1.78	70	4.85	3.76	6	1.62	63	4.85	3.77	6
ATP30	209.40	181	10.17	7.90	10	210.42	179	10.17	7.90	10
ATP31	1001.43	268	15.75	13.19	**15**	925.57	253	15.75	13.19	16
ATP32	194.24	297	13.90	11.84	14	184.02	283	13.90	11.82	14
ATP33	123.94	213	13.89	11.61	15	120.10	212	13.89	11.62	**14**
ATP34	91.68	106	6.23	5.14	7	71.98	94	6.23	5.14	**6**
ATP35	121.90	110	7.85	7.07	8	110.22	108	7.85	7.07	8
ATP36	54.06	124	9.69	7.13	9	61.71	139	9.71	7.13	9
ATP37	228.88	189	13.63	10.68	13	259.70	205	13.63	10.67	**12**
ATP38	280.36	187	11.37	9.82	**11**	266.16	198	11.37	9.82	12
ATP39	79.03	122	12.68	10.81	13	87.09	135	12.68	10.81	13
CHL1s	6.21	110	6.86	5.08	7	6.64	117	6.86	5.07	7
CHL2s	0.09	22	2.45	2.25	3	0.09	22	2.45	2.25	3
CHL5	0.03	14	3.45	2.60	3	0.03	14	3.45	2.60	3
CHL6	9.38	127	5.79	4.88	7	9.86	131	5.79	4.88	**6**
CHL7	9.58	146	5.99	5.17	7	9.73	153	5.99	5.17	7
CU1	214.30	393	41.86	37.16	**44**	213.2	394	41.86	37.16	45
CU10	25.30	130	15.77	13.72	16	23.36	128	15.77	13.72	16
CU11	188.27	210	14.27	12.50	15	189.14	212	14.27	12.50	15
CU2	2.67	81	14.78	13.56	16	3.11	82	14.78	13.56	16
CU3	6.91	123	22.21	19.92	23	6.93	126	22.21	19.91	23
CU4	16.12	126	17.28	15.65	18	15.48	128	17.28	15.65	18
CU5	14.36	135	21.52	19.34	23	18.3	140	21.52	19.38	**22**
CU6	9.67	121	19.99	17.36	20	10.95	128	19.99	17.35	20
CU7	1.69	64	10.40	9.14	11	1.60	61	10.40	9.14	11
CU8	4.46	98	15.31	13.49	16	3.87	88	15.31	13.50	16
CU9	1.83	63	15.08	13.22	15	1.64	62	15.08	13.22	15

Table 5.3: Results of DP-Generator with and without the symmetry breaking constraint, part 1.

In all tables, "t" refers to time in seconds, "Cls" refers to the number of columns added, "LP" is the solution to the LP relaxation, while "LP_{CG}" is the solution to the LP relaxation using the added columns. Finally, "Sol" is the integral solution.
In Table 5.3 and 5.4 we can see the results of the column generation using DP-Generator as a strip generator. In the early stages of testing, the symmetry breaking constraints 3.4 from section 3.2, seemed to hinder the integrality heuristic, promoting infeasible solutions. The reason behind this is because the integrality heuristic makes each strip integral step-wise. In cases where a sheet might have more parts that are fractional, and hence cut, as the following sheet, an infeasibility happens (due to the ordering of the sheet patterns by height). As such, a new ordering is implemented when the algorithm is run without the symmetry breaking constraints.

$$y_j \leq y_{j-1} \qquad \forall j = 2, \ldots, n$$

referred to as $soft$ symmetry breaking constraints. Although it seems the symmetry breaking constraints perform better most of the time, there are instance where the inverse holds true, as well.

In Table 5.5 and 5.6 the results for the Heuristic-Generator 1 are shown. The variable *heuristic* in this context is set to 1 and to 5, meaning the amount of heights that are checked before the best is returned. The run time for cases when $heuristic = 5$ can increase substantially, as seen in the ATP instances, although usually it does not make a large difference. The LP_{CG} for $heuristic = 5$ is always at least as good as for $heuristic = 1$, although this does not always translate to a better overall solution. Overall the LP_{GC} is competitive with DP-Generator, beating it especially in computing time.

For the Heuristic-Generator 2 algorithm, the LP relaxed solutions are competitive with the Heuristic-Generator 1. However, both for the run time and for the integral solutions, the results vary. Interestingly, the number of columns added by Heuristic-Generator 2 is not significantly greater than for Heuristic-Generator 1.

The final results can be compared in Table 5.8. The dynamic programming implementation used here is an adaption from the SLOPP solution presented by Cintra et al. [CMWX08]. DP-Imp performs better than dynamic programming in most cases. The exception here are the randomly generated $UU7s$, $UU8s$, and $UU9s$ instances, where DP-Generator does not perform well. In the case of the insertion heuristic, however, results vary. There are many cases in which DP-Generator finds a better solution, however there are cases where the insertion heuristic is much better.

Figure 5.1 and Figure 5.2 show the average gap between the integral solution *Sol* and LP_{GC} for all 3-staged variants. DP-0 refers to DP-Generator with symmetry= 0, DP-1 to DP-Generator with symmetry= 1, Heur-1 to Heuristic-Generator with heuristic= 1, Heur-5 to Heuristic-Generator with heuristic= 5, and finally Heur2 to Heuristic-Generator 2. The two graphs are separated in the same manner as the tables, to keep consistency. Moreover, the instances are separated by difficulty this way as well. The gap shows

	Symmetry = 0					Symmetry = 1				
Instance	t	Cls	LP	LP_{CG}	Sol	t	Cls	LP	LP_{CG}	Sol
Hchl3s	0.86	28	3.00	2.67	4	0.67	27	3.00	2.67	4
Hchl4s	0.46	27	2.65	1.75	2	0.46	27	2.65	1.75	2
Hchl5s	10.31	96	4.02	3.40	5	10.55	97	4.02	3.40	5
Hchl6s	4.73	73	4.97	4.32	**5**	5.29	80	4.97	4.32	6
Hchl7s	39.34	182	7.76	6.44	9	42.38	191	7.76	6.44	**8**
Hchl8s	0.12	20	1.40	1.24	2	0.12	20	1.40	1.24	2
Hs	0.06	23	16.48	15.44	18	0.10	21	16.48	15.44	18
HZ1s	0.10	9	28.90	28.68	**33**	0.09	9	28.90	28.68	34
M1	0.45	79	76.58	72.36	**84**	0.42	77	76.58	72.36	85
M2	0.58	45	61.87	61.06	66	0.56	41	61.87	61.06	66
M3	0.64	112	84.08	75.11	91	0.67	116	84.08	75.11	91
M4s	0.56	92	76.53	72.41	84	0.51	80	76.53	72.41	**80**
M5	0.61	87	79.02	74.14	92	0.66	89	79.02	74.14	**88**
OF1	0.08	24	3.68	3.21	5	0.07	20	3.68	3.21	5
OF2	0.05	22	4.32	3.44	5	0.05	24	4.32	3.44	5
STS2s	1.96	89	12.56	11.19	13	2.32	81	12.56	11.20	13
STS4s	1.42	59	5.06	4.54	6	1.72	69	5.06	4.54	6
UU10	412.05	872	517.69	483.04	**574**	409.85	881	517.69	483.04	587
UU1s	5.13	195	304.46	293.12	362	7.37	246	304.46	293.05	**356**
UU2s	18.38	280	300.00	283.86	356	18.19	273	300.00	283.86	**334**
UU3s	16.49	395	247.56	231.86	297	24.53	397	247.56	231.86	**292**
UU4s	57.25	516	372.39	352.16	467	54.25	502	372.39	352.26	**450**
UU5s	187.89	618	494.33	468.35	603	205.12	662	494.33	468.02	**589**
UU6s	48.86	494	393.04	378.55	477	52.41	494	393.04	378.55	**459**
UU7s	374.45	699	478.99	442.05	547	329.81	727	478.99	442.06	**536**
UU8s	305.77	771	550.72	522.18	658	320.87	800	550.74	522.18	**642**
UU9	547.42	865	656.04	618.78	822	406.47	885	656.04	618.78	**808**
W	0.64	63	18.58	17.16	23	0.57	58	18.58	17.16	23

Table 5.4: Results of DP-Generator with and without the symmetry breaking constraint, part 2.

	Heuristic = 1					Heuristic = 5				
Instance	t	Cls	LP	LP_{CG}	Sol	t	Cls	LP	LP_{CG}	Sol
2s	0.02	17	2.29	1.65	3	0.04	14	2.29	1.65	3
3s	0.06	3	20.94	20.86	28	0.08	3	20.94	20.86	28
A1s	0.13	52	19.78	17.98	24	0.61	154	19.78	17.98	24
A2s	0.11	64	11.47	10.47	13	0.53	105	11.47	10.47	13
A3	0.16	78	7.79	6.81	9	3.11	415	7.79	6.81	**8**
A4	0.23	70	4.34	4.03	5	0.53	76	4.34	4.03	5
A5	0.45	84	4.85	3.75	6	0.86	88	4.85	3.75	6
ATP30	28.06	289	10.17	7.90	10	273.16	485	10.17	7.90	10
ATP31	78.38	489	15.75	13.19	15	223.12	745	15.75	13.19	15
ATP32	227.96	1652	13.90	11.78	14	755.23	2079	13.90	11.78	14
ATP33	75.27	615	13.89	11.60	14	240.11	955	13.89	11.60	14
ATP34	15.67	242	6.23	5.13	6	67.09	401	6.23	5.13	6
ATP35	19.44	174	7.85	7.07	9	51.65	385	7.85	7.07	9
ATP36	8.09	178	9.69	7.11	10	34.25	950	9.69	7.11	**9**
ATP37	174.9	1623	13.63	10.66	13	388.76	1107	13.63	10.66	13
ATP38	35.02	295	11.37	9.82	12	254.46	908	11.37	9.82	12
ATP39	10.82	228	12.68	10.77	13	24.07	331	12.68	10.77	13
CHL1s	1.28	174	6.86	5.06	7	1.99	136	6.86	5.06	7
CHL2s	0.05	24	2.45	2.25	3	0.08	28	2.45	2.25	3
CHL5	0.03	34	3.45	2.60	**3**	0.47	283	3.45	2.60	4
CHL6	1.49	177	5.79	4.88	6	2.78	161	5.79	4.88	6
CHL7	1.67	251	5.99	5.15	7	4.04	201	5.99	5.15	7
CU1	23.66	511	41.86	37.13	45	36.85	489	41.86	37.12	**44**
CU10	3.71	156	15.77	13.72	16	4.09	150	15.77	13.72	16
CU11	19.76	285	14.27	12.47	16	32.43	271	14.27	12.47	**15**
CU2	0.57	129	14.78	13.52	16	0.93	99	14.78	13.52	16
CU3	1.04	164	22.21	19.85	24	2.02	163	22.21	19.85	**23**
CU4	2.44	150	17.28	15.65	19	2.69	135	17.28	15.65	**18**
CU5	1.19	168	21.52	19.29	**22**	10.79	295	21.52	19.29	23
CU6	1.76	152	19.99	17.26	20	1.81	131	19.99	17.26	20
CU7	0.45	84	10.40	9.14	11	0.56	77	10.40	9.14	11
CU8	0.38	101	15.31	13.48	15	0.99	102	15.31	13.47	15
CU9	0.33	83	15.08	13.22	16	0.42	66	15.08	13.22	16

Table 5.5: Results of Heuristic-Generator 1 generating 1 and 5 different heights, part 1.

	Heuristic = 1					Heuristic = 5				
Instance	t	Cls	LP	LP_{CG}	Sol	t	Cls	LP	LP_{CG}	Sol
Hchl3s	0.33	38	3.00	2.67	4	0.27	28	3.00	2.67	4
Hchl4s	0.20	36	2.65	1.75	2	0.29	34	2.65	1.75	2
Hchl5s	2.24	149	4.02	3.36	5	3.67	142	4.02	3.36	5
Hchl6s	0.95	128	4.97	4.26	**5**	2.12	122	4.97	4.26	6
Hchl7s	5.61	277	7.76	6.40	8	9.38	261	7.76	6.40	8
Hchl8s	0.08	35	1.40	1.21	2	0.14	26	1.40	1.21	2
Hs	0.05	22	16.48	15.44	**17**	0.07	21	16.48	15.44	18
HZ1s	0.09	12	28.90	28.68	**32**	0.12	17	28.90	28.68	35
M1	0.29	86	76.58	72.36	83	0.37	126	76.58	72.36	**81**
M2	0.49	57	61.87	61.06	**66**	0.67	55	61.87	61.06	67
M3	0.54	137	84.08	75.11	89	0.43	136	84.08	75.11	**88**
M4s	0.39	82	76.53	72.41	84	0.36	114	76.53	72.41	**82**
M5	0.33	109	79.02	74.14	**89**	0.39	105	79.02	74.14	94
OF1	0.03	26	3.68	3.24	4	0.08	30	3.68	3.26	4
OF2	0.03	26	4.32	3.43	5	0.05	29	4.32	3.43	5
STS2s	0.32	119	12.56	11.09	13	0.92	98	12.56	11.09	13
STS4s	0.51	80	5.06	4.53	6	0.75	74	5.06	4.53	6
UU10	125.65	1149	517.69	482.91	589	97.44	1119	517.69	482.74	**581**
UU1s	3.27	307	304.46	293.08	**359**	2.59	249	304.46	293.07	361
UU2s	8.72	394	300.00	283.86	**348**	5.43	294	300.00	283.86	349
UU3s	7.90	570	247.56	231.86	290	4.81	488	247.56	231.86	**287**
UU4s	19.23	733	372.39	350.28	**451**	11.64	623	372.39	350.23	460
UU5s	44.64	1001	494.33	466.31	**591**	58.18	1203	494.33	466.31	599
UU6s	15.97	722	393.04	378.55	495	27.36	736	393.04	378.55	**489**
UU7s	50.71	998	478.99	442.04	**553**	45.15	924	478.99	442.04	557
UU8s	67.77	1068	550.72	522.18	**665**	294.40	2506	550.72	522.18	667
UU9	110.47	1226	656.04	618.78	803	139.07	1068	656.04	618.78	**789**
W	0.16	63	18.58	17.16	24	0.26	57	18.58	17.16	24

Table 5.6: Results of Heuristic-Generator 1 generating 1 and 5 different heights, part 2.

	Heuristic-Generator 2									
Instance	t	Cls	LP	LP_{CG}	Sol	Instance	t	Cls	LP	LP_{CG}	Sol
2s	0.02	20	2.29	1.65	2	CU8	0.76	131	15.31	13.47	16
3s	0.05	5	20.94	20.86	28	CU9	0.41	81	15.08	13.22	15
A1s	0.08	57	19.78	17.98	25	Hchl3s	0.37	37	3.00	2.66	4
A2s	0.14	91	11.47	10.47	13	Hchl4s	0.23	36	2.65	1.75	2
A3	0.10	82	7.79	6.81	8	Hchl5s	2.47	203	4.02	3.36	4
A4	0.23	81	4.34	4.02	5	Hchl6s	1.04	134	4.97	4.27	6
A5	0.36	104	4.85	3.75	6	Hchl7s	5.55	351	7.76	6.40	9
ATP30	46.81	338	10.17	7.90	10	Hchl8s	0.07	41	1.40	1.20	2
ATP31	335.79	448	15.75	13.19	16	Hs	0.05	26	16.48	15.44	17
ATP32	827.55	1732	13.90	11.80	15	HZ1s	0.09	16	28.90	28.68	33
ATP33	23.96	610	13.89	11.6	14	M1	0.34	109	76.58	72.36	79
ATP34	51.36	723	6.23	5.13	7	M2	0.65	53	61.87	61.06	66
ATP35	23.63	287	7.85	7.07	9	M3	0.54	143	84.08	75.11	87
ATP36	7.24	221	9.69	7.11	10	M4s	0.40	117	76.53	72.41	81
ATP37	153.52	746	13.63	10.66	13	M5	0.53	118	79.02	74.14	92
ATP38	31.93	297	11.37	9.82	12	OF1	0.04	24	3.68	3.25	5
ATP39	12.15	260	12.68	10.77	13	OF2	0.03	28	4.32	3.43	5
CHL1s	1.39	189	6.86	5.06	8	STS2s	0.42	153	12.56	11.09	13
CHL2s	0.04	28	2.45	2.25	3	STS4s	0.55	88	5.06	4.53	6
CHL5	0.03	32	3.45	2.60	3	UU10	150.96	1370	517.69	482.79	572
CHL6	3.02	197	5.79	4.88	6	UU1s	2.60	303	304.46	293.12	363
CHL7	1.24	246	5.99	5.15	7	UU2s	6.63	420	300.00	283.86	344
CU1	20.06	608	41.86	37.13	44	UU3s	12.63	498	247.56	231.86	295
CU10	4.54	192	15.77	13.73	16	UU4s	12.51	804	372.39	350.31	461
CU11	15.25	315	14.27	12.47	15	UU5s	45.74	1074	494.33	466.31	600
CU2	0.70	135	14.78	13.52	16	UU6s	24.00	774	393.04	378.55	475
CU3	1.17	236	22.21	19.86	24	UU7s	48.54	1119	478.99	442.04	571
CU4	1.95	194	17.28	15.65	18	UU8s	71.76	1261	550.72	522.18	660
CU5	1.58	213	21.52	19.29	22	UU9	148.37	1245	656.04	618.78	820
CU6	2.47	188	19.99	17.26	20	W	0.28	79	18.58	17.16	24
CU7	0.45	106	10.40	9.14	11						

Table 5.7: Results of Heuristic-Generator 2

Instance	InsHeur	DynProg	DP-Gen	Instance	InsHeur	DynProg	DP-Gen
2s	3	3	3	CU8	16	**15**	16
3s	**23**	28	26	CU9	16	**15**	**15**
A1s	**23**	26	25	Hchl3s	4	4	4
A2s	**12**	12	13	Hchl4s	3	3	**2**
A3	8	8	8	Hchl5s	5	**4**	5
A4	**5**	7	**5**	Hchl6s	6	6	6
A5	6	**5**	6	Hchl7s	8	8	8
ATP30	11	10	10	Hchl8s	2	2	2
ATP31	16	16	16	Hs	18	**17**	18
ATP32	14	14	14	HZ1s	**30**	31	34
ATP33	14	14	14	M1	82	**80**	85
ATP34	7	7	**6**	M2	67	74	**66**
ATP35	**8**	9	**8**	M3	**90**	100	91
ATP36	10	10	**9**	M4s	82	**80**	**80**
ATP37	14	**12**	**12**	M5	**85**	108	88
ATP38	12	12	12	OF1	**4**	**4**	5
ATP39	13	13	13	OF2	5	5	5
CHL1s	7	**6**	7	STS2s	13	**12**	13
CHL2s	3	3	3	STS4s	6	6	6
CHL5	4	4	**3**	UU10	**540**	602	587
CHL6	6	6	6	UU1s	**326**	387	356
CHL7	7	7	7	UU2s	**312**	342	334
CU1	**13**	**13**	14	UU3s	**258**	260	292
CU10	17	**16**	**16**	UU4s	**398**	456	450
CU11	15	15	15	UU5s	**527**	542	589
CU2	16	**15**	16	UU6s	**410**	424	459
CU3	24	**21**	23	UU7s	**499**	515	536
CU4	18	**17**	18	UU8s	**580**	625	642
CU5	23	**22**	**22**	UU9	**695**	773	808
CU6	21	**19**	20	W	24	28	**23**
CU7	11	**10**	11				

Table 5.8: Solutions from Dynamic Programming, the Insertion Heuristic and DP-Generator with symmetry = 1

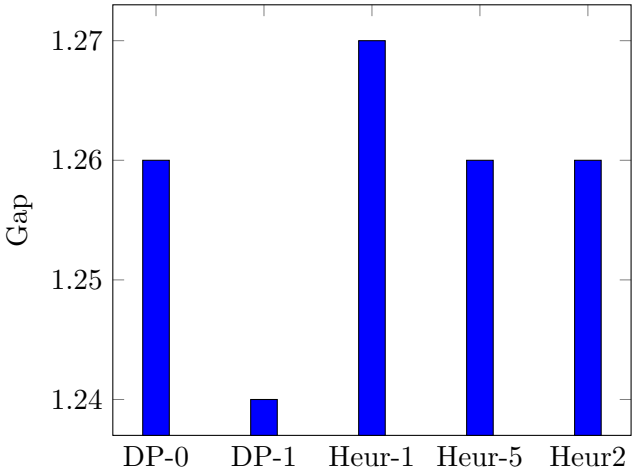


Figure 5.1: Average gap between Sol and LP_{GC} (Sol/LP_{GC}) for all 3-staged variants on instances up to CU9.

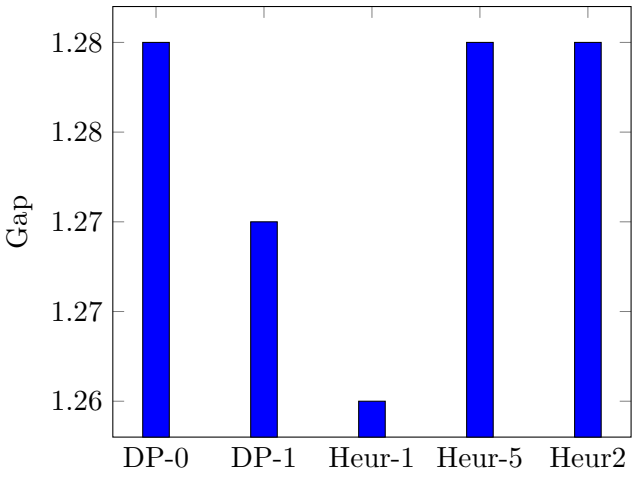
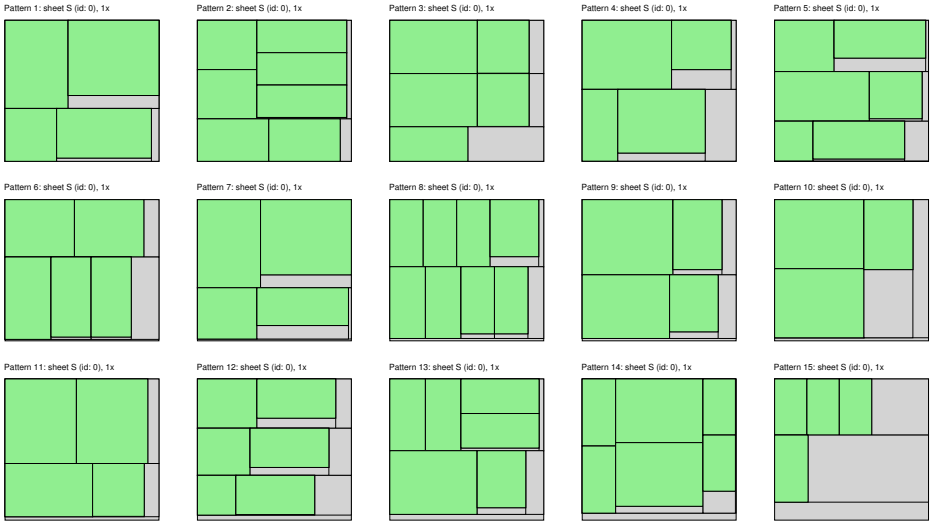


Figure 5.2: Average gap between Sol and LP_{GC} (Sol/LP_{GC}) for all 3-staged variants on instances from Hchl3s.

that on average, using the symmetry constraints consistently outperforms DP-Generator without the symmetry constraints. It also appears that the more random nature of using heuristic= 1 performs best on more complex instances, while performing worst on the smaller, easier instances. This suggests that choosing the strip with the *least* negative reduced cost plays a factor in easier instances, but is a potential weakness in more complex instances. The differences are marginal, however. There seems to be no difference between Heuristic-Generator with heuristic= 5 and Heuristic-Generator 2. The conceptual difference is that Heuristic-Generator 2 generates multiple strips per set of dual variables, whereas Heuristic-Generator only chooses the best (out of 5, in case of heuristic= 5). There is therefore a higher likelihood that they share similar strips.
Figure 5.3 shows two solutions to the instance CU9. Figure 5.3a is the solution from DP-Generator with symmetry constraints, whereas Figure 5.3b is the solution from the Insertion Heuristic. In the solution of DP-Generator, Pattern 14 (second to last) is of note. It consists only of one large strip, almost the size of the sheet itself.



(a) Solutiont to instance CU9 using DP-Generator, with symmetry = 1.

(b) Solution to instance CU9 using the Insertion Heuristic.

Figure 5.3: Solutions to instance CU9

CHAPTER 6

Conclusions and Future Work

Column generation based on strip generation clearly has potential, as can be seen in the LP relaxed solutions. In some cases, the rather naive integrality heuristic manages to beat more classical approaches, such as dynamic programming or an insertion heuristic. It does not consistently beat them, however, even with substantially longer run times.

The strip generating algorithms offer expected solutions. They reach the LP relaxed optimum in most cases, including both heuristics. However, results start to spread concerning integral solutions. Although, for example, Heuristic-Generator 1 offers similar LP relaxed solutions, the integrality heuristic can not translate that into a solution as good as with DP-Generator. This suggests that the strips generated heuristically are only used fractionally by the master problem, where $x_{pj} < 1$. The gaps between the integral solution and the LP relaxed solutions give a good insight into the importance of choosing the strip with the most negative reduced cost, versus choosing the first random strip with negative reduced cost. In particular, it seems dependent on the instance which generator should be used. The larger and more complex the instance, the more random the choice of strip appears to work better.
In the case of $K = 2$, strip generation does not fare so well. The number of available strips that can be generated is too small to take advantage of the column generation framework. This means a lot of computing time solving the initial ILP is wasted. The strip generators in this case do not allow trimming, i.e. using a smaller element in a larger strip, therefore requiring to trim the wasted material. This severely limits the number of possible strips, and plays a role in its performance.

The overall run time of the SSCG is not a problem, whereas the space requirements are. Cplex manages to find solutions using up to around 6000 sheets. Instances that require more sheets cannot be solved, and throw memory exceptions. Although this could be managed with a larger allotment of memory, the run times in these cases also

become increasingly large. It stands to reason That the SSCG can realistically handle no more than 5000 sheets. This is because although we see solutions with almost 6000 sheets, these are in cases of few added columns, where the model remains an equal size throughout the entire run.

Future Work

Given the integral results, the most potential seems to be in improving the integrality heuristic. This could be done using a more refined heuristic, such as making elements integral, as opposed to strips, and repacking them. Branch and Price is an approach that promises even better results, as it can be used to find the optimal integral solution. Both approaches demand significant computing time, however. In any case, it appears to be worth exploring further.
Another avenue worth exploring might be allowing trimming in the case of $K = 2$, and reevaluating the results. It is expected that the increased number in potential strips produces better results.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ANW14] Mohsen Afsharian, Ali Niknejad, and Gerhard Wäscher. A heuristic, dynamic programming-based approach for a two-dimensional cutting problem with defects. *OR Spectrum*, 36(4):971–999, 2014.

[APR00] Rumen Andonov, Vincent Poirriez, and Sanjay Rajopadhye. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2):394–407, 2000.

[AVPT02] Ramon Alvarez-Valdes, Antonio Parajon, and Jose M. Tamarit. A computational study of LP-based heuristic algorithms for two-dimensional guillotine cutting stock problems. *OR Spectrum*, 24(2):179–192, 2002.

[Bea85] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *The Journal of the Operational Research Society*, 36(4):297–306, 1985.

[BW87] J. O. Berkey and P. Y. Wang. Two-Dimensional Finite Bin-Packing Algorithms. *The Journal of the Operational Research Society*, 38(5):423–429, 1987.

[CGJ82] Fan RK Chung, Michael R Garey, and David S Johnson. On packing two-dimensional bins. *SIAM Journal on Algebraic Discrete Methods*, 3(1):66–76, 1982.

[CGJT80] Edward G Coffman, Jr, Michael R Garey, David S Johnson, and Robert Endre Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[CHC00] V-D Cung, Mhand Hifi, and B Cun. Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm. *International Transactions in Operational Research*, 7(3):185–210, 2000.

[CHH04] Y Cui, L Huang, and D He. Generating optimal multiple-segment cutting patterns for rectangular blanks. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 218(11):1483–1490, 2004.

[CJGJ96] Edward G Coffman Jr, Michael R Garey, and David S Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[CL08] Yaodong Cui and Zhiyong Liu. T-shape homogenous block patterns for the two-dimensional cutting problem. *Journal of Global Optimization*, 41(2):267–281, 2008.

[CMWX08] G.F. Cintra, F.K. Miyazawa, Y. Wakabayashi, and E.C. Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191(1):61–85, 2008.

[Cui13] Yaodong Cui. A new dynamic programming procedure for three-staged cutting patterns. *Journal of Global Optimization*, 55(2):349–357, 2013.

[CWL05] Y Cui, Z Wang, and J Li. Exact and heuristic algorithms for staged cutting problems. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 219(2):201–207, 2005.

[DR14] Frederico Dusberger and Günther R. Raidl. A Variable Neighborhood Search Using Very Large Neighborhood Structures for the 3-Staged 2-Dimensional Cutting Stock Problem. In Maria J. Blesa, Christian Blum, and Stefan Voß, editors, *Hybrid Metaheuristics*, volume 8457 of *LNCS*, pages 85–99. Springer, 2014.

[DR15a] Frederico Dusberger and Günther R. Raidl. A Scalable Approach for the K-Staged Two-Dimensional Cutting Stock Problem with Variable Sheet Size. In *Computer Aided Systems Theory - EUROCAST 2015 - 15th International Conference, Las Palmas de Gran Canaria, Spain, February 8-13, 2015, Revised Selected Papers*, pages 384–392, 2015.

[DR15b] Frederico Dusberger and Günther R. Raidl. Solving the 3-staged 2-dimensional cutting stock problem by dynamic programming and variable neighborhood search. In *The 3rd International Conference on Variable Neighborhood Search (VNS'14)*, volume 47 of *Electronic Notes in Discrete Mathematics*, pages 133–140. Elsevier, 2015.

[DW60] George B. Dantzig and Philip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, 1960.

[Dyc90] Harald Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[FG87] J B Galamlx Frenk and Gábor Galambos. Hybrid next-fit algorithm for the two-dimensional rectangle bin-packing problem. *Computing*, 39(3):201–217, 1987.

[FHZ98] D Fayard, M Hifi, and V Zissimopoulos. An efficient approach for large-scale two-dimensional guillotine cutting stock problems. *Journal of the Operational Research Society*, pages 1270–1277, 1998.

[Fle13] Krzysztof Fleszar. Three Insertion Heuristics and a Justification Improvement Heuristic for Two-Dimensional Bin Packing with Guillotine Cuts. *Computers & Operations Research*, 40(1):463–474, 2013.

[GG61] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9(6):849–859, 1961.

[GG63] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting Stock Problem—Part II. *Operations Research*, 11(6):863–888, 1963.

[GG65] P. C. Gilmore and R. E. Gomory. Multistage Cutting Stock Problems of Two and More Dimensions. *Operations Research*, 13(1):94–120, 1965.

[GG66] PC Gilmore and RE Gomory. The theory and computation of knapsack functions. *Operations Research*, 14(6):1045–1074, 1966.

[GGJY76] Michael R Garey, Ronald L Graham, David S Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A*, 21(3):257–298, 1976.

[Glo97] Fred Glover. Tabu search and adaptive memory programming — advances, applications and challenges. In RichardS. Barr, RichardV. Helgason, and JefferyL. Kennington, editors, *Interfaces in Computer Science and Operations Research*, volume 7 of *Operations Research/Computer Science Interfaces Series*, pages 1–75. Springer US, 1997.

[Her72] J. C. Herz. Recursive Computational Procedure for Two-dimensional Stock Cutting. *IBM Journal of Research and Development*, 16(5):462–469, 1972.

[HM05] Mhand Hifi and Rym M'Hallah. An Exact Algorithm for Constrained Two-Dimensional Two-Staged Cutting Problems. *Operations Research*, 53(1):140–150, 2005.

[HZ96] Mhand Hifi and Vassilis Zissimopoulos. Une amélioration de l'algorithme récursif de herz pour le problème de découpe à deux dimensions. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle*, 30(2):111–125, 1996.

[HZL+14] Shaohui Hong, Defu Zhang, Hoong Chuin Lau, XiangXiang Zeng, and Yain-Whar Si. A hybrid heuristic algorithm for the 2D variable-sized bin packing problem. *European Journal of Operational Research*, 238(1):95–103, oct 2014.

[JDU+74] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[KP04] Hans Kellerer and Ulrich Pferschy. Improved dynamic programming in connection with an fptas for the knapsack problem. *Journal of Combinatorial Optimization*, 8(1):5–11, 2004.

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Knapsack problems. 2004.

[Law77] E.L. Lawler. Fast approximation algorithms for knapsack problems. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 206–213, Oct 1977.

[LMM02] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[LMV02] Andrea Lodi, Silvano Martello, and Daniele Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1–3):379 – 396, 2002.

[LMV04] Andrea Lodi, Silvano Martello, and Daniele Vigo. Models and bounds for two-dimensional level packing problems. *Journal of Combinatorial Optimization*, 8:363–379, 2004.

[MAA92] RN Morabito, MN Arenales, and VF Arcaro. An and—or-graph approach for two-dimensional cutting problems. *European Journal of Operational Research*, 58(2):263–271, 1992.

[MAS] S. Gupta M. Akella and A. Sarkar. Branch and price, column generation for solving huge ips. `http://www.acsu.buffalo.edu/~nagi/courses/684/price.pdf`, University at Buffalo. Lecture Slides.

[MJ07] Laura A. McLay and Sheldon H. Jacobson. Algorithms for the bounded set-up knapsack problem. *Discrete Optimization*, 4(2):206 – 212, 2007.

[MP10] Reinaldo Morabito and Vitória Pureza. A heuristic approach based on dynamic programming and and/or-graph search for the constrained two-dimensional guillotine cutting problem. *Annals of Operations Research*, 179(1):297–315, 2010.

[MT77] Silvano Martello and Paolo Toth. Branch-and-bound algorithms for the solution of the general unidimensional knapsack problem. *Advances in Operations Research, North-Holland, Amsterdam*, pages 295–301, 1977.

[MT90a] Silvano Martello and Paolo Toth. An exact algorithm for large unbounded knapsack problems. *Oper. Res. Lett.*, 9(1):15–20, January 1990.

[MT90b] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[MT06] Michele Monaci and Paolo Toth. A Set-Covering-Based Heuristic Approach for Bin-Packing Problems. *INFORMS Journal on Computing*, 18(1):71–85, 2006.

[MV98] Silvano Martello and Daniele Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Manage. Sci.*, 44(3):388–399, 1998.

[OF90] JoseFernando Oliveira and JoseSoeiro Ferreira. An improved version of wang's algorithm for two-dimensional cutting problems. *European Journal of Operational Research*, 44(2):256–266, 1990.

[OF94] J. F. Oliveira and J. S. Ferreira. A faster variant of the Gilmore and Gomory technique for cutting stock problems. *Belgian Journal of. Operational Research, Statistics and Computer Science*, 34(1):23–38, 1994.

[Pfe99] U. Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63(4):419–430, 1999.

[PR04] Jakob Puchinger and Günther R. Raidl. An Evolutionary Algorithm for Column Generation in Integer Programming: An Effective Approach for 2D Bin Packing. In Xin Yao and et al., editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 642–651. Springer, 2004.

[PR07] Jakob Puchinger and Günther R. Raidl. Models and algorithms for three-stage two-dimensional bin packing . *European Journal of Operational Research*, 183(3):1304–1327, 2007.

[PRK04] Jakob Puchinger, Günther R. Raidl, and Gabriele Koller. Solving a Real-World Glass Cutting Problem. In Jens Gottlieb and Günther R. Raidl, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 3004 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2004.

[PSWB11] Jakob Puchinger, PeterJ. Stuckey, MarkG. Wallace, and Sebastian Brand. Dantzig-Wolfe decomposition and branch-and-price solving in G12. *Constraints*, 16(1):77–99, 2011.

[Sch97] G. Scheithauer. Equivalence and Dominance for Problems of Optimal Packing of Rectangles. *Ricerca Operativa*, 83:3–34., 1997.

[Wan83] PY Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Operations Research*, 31(3):573–586, 1983.

[WHS07] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.