Die approbierte Originalversion dieser Diplom-/ Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Branch-and-Bound Approach for the Constrained K-Staged 2-Dimensional Cutting Stock Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Bernhard Bonigl, BSc
Registration Number 0926003

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 16th June, 2015

Bernhard Bonigl   Günther R. Raidl

Erklärung zur Verfassung der Arbeit

Bernhard Bonigl, BSc
Goldeggerstraße 5, 3100 St.Pölten

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, 16. Juni 2015

____
Bernhard Bonigl

Acknowledgements

I want to thank my advisors Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl and Univ.Ass. Dipl.-Ing. Frederico Dusberger for supervising my master thesis. I would like to put special emphasis on all the help and feedback Frederico provided, adding ideas and references from the literature, allowing me to progress smoothly with my thesis.

I want to thank my parents for their continuous support. Without them my studies would not have been possible.

Finally I want to express my gratitude to my partner and my remaining family, who provided moral support and never doubted me.

Abstract

The K-staged two-dimensional cutting stock problem with variable sheet size (K-2CSV) represents a common problem in industry where a cutting pattern to cut rectangular shaped elements out of large stock sheets is required. The NP-hard nature of the problem makes it difficult to find a good pattern, which directly translates to unused waste material of the stock sheet. The historical approach is to try and find an optimal pattern through dynamic programming, which later was supplemented by Branch-and-Bound and heuristic approaches.

In this paper we develop a bottom-up Branch-and-Bound approach, creating an optimal cutting pattern for a singular stock sheet. While top-down approaches sucessively subdivide the sheet into smaller rectangles and ultimately into the required elements, bottom-up approaches combine elements and combinations thereof together to create whole patterns.

The process is supplemented with a general framework to integrate it into the implementation of the *Algorithms and Complexity Group* at the TU Wien. The framework allows to solve problem instances spanning multiple stock sheets by applying the algorithm multiple times.

We then boost the performance of the algorithm by improving the used lower and upper bounds as well as reducing the search space through the detection of dominated or duplicate patterns.

Lastly, using different settings we apply the algorithm to problem instances taken from the literature to obtain computational results.

Kurzfassung

Das K-stufige 2-dimensionale Zuschnittproblem mit variabler Plattengröße (*K-2CSV*) ist ein häufig in der Industrie auftretendes Problem. Oftmals sollen rechteckige Elemente aus großen Platten oder Blättern herausgeschnitten werden. Die NP-harte Natur dieses Problems macht es schwer ein gutes Schnittmuster zu finden das möglichst wenig Ausschuss produziert. Der historische Ansatz ist mittels Dynamic Programming ein optimales Muster zu finden. Später wurde diese Vorgehensweise durch Branch-and-Bound und heuristische Ansätze ergänzt.

In dieser Arbeit entwickeln wir einen bottom-up Branch-and-Bound Ansatz welcher ein optimales Schnittmuster für eine einzelne Platte aus dem Bestand berechnet. Während ein top-down Ansatz die Platte Schritt für Schritt in kleinere Rechtecke unterteilt um letztendlich an die geforderten Elemente zu gelangen, kombiniert ein bottom-up Ansatz Elemente und Kombinationen von Elementen um sein Ziel zu erreichen.

Dieser Prozess wird um ein generelles Framework erweitert um es in die Implementierung der *Algorithms and Complexity Group* der TU Wien zu integrieren. Das Framework erlaubt das Lösen von Probleminstanzen die mehrere Platten benötigen indem es den Algorithmus auf jede Platte einzeln anwendet.

Anschließend verbessern wir die Leistung des Algorithmus indem wir bessere Schranken finden und den Suchraum durch Erkennung von dominierten oder doppelten Mustern einschränken.

Zu guter Letzt wird der Algorithmus in verschiedenen Konfigurationen auf Probleminstanzen aus der Literatur angewandt um an rechnerische Ergebnisse zu gelangen.

Contents

Abstract vii

Kurzfassung ix

Contents xi

List of Figures xii

List of Tables xii

List of Algorithms xiii

1 Introduction 1
- 1.1 Aim of This Work . . . 1
- 1.2 Guillotine Cuts . . . 2
- 1.3 Problem Definition . . . 3
- 1.4 Cutting Tree . . . 5

2 Literature Survey 9

3 Solving Single Sheets by Branch-and-Bound 13
- 3.1 Existing K-2CSV Framework . . . 13
- 3.2 Branch-and-Bound Framework . . . 14
- 3.3 Simple Algorithm . . . 15
- 3.4 CSA Algorithm . . . 19
- 3.5 Algorithm Improvements . . . 22
- 3.6 Flexible Algorithm . . . 25

4 Computational Results 27
- 4.1 Instances . . . 28
- 4.2 Results . . . 29
- 4.3 Restricted Stages . . . 38
- 4.4 Comparison to a Beam-Search Approach Utilizing an Insertion Heuristic . 46
- 4.5 Comparison of the Results . . . 48

5 Summary, Conclusion and Future Work **51**
5.1 Future Work . . . 52

Bibliography **55**

List of Figures

1.1 Guillotine Cuts . . . 3
1.2 Non-Guillotine Cuts . . . 3
1.3 Guillotine Cut Stages . . . 3
1.4 A Three-Staged Cutting Tree and the Corresponding Pattern . . . 6

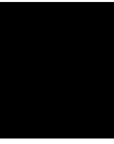
3.1 Strip Partitioning of Unused Pattern Space . . . 22

List of Tables

4.1 Score, Stage and Timeout Results for K=*9* . . . 30
4.2 Runtime Results for K=*9* . . . 31
4.3 Runtime Until Best Solution Found for K=*9* . . . 33
4.4 Branches Explored for K=*9* . . . 35
4.5 Leaf Nodes Evaluated for K=*9* . . . 37
4.6 Runtime Results for K=*6* . . . 38
4.7 Runtime Results for K=*4* . . . 40
4.8 Runtime Results for K=*3* . . . 42
4.9 Runtime Results For K=*2* . . . 44
4.10 Beam-Search Comparison . . . 46
4.11 Comparison of Score Gaps to Optimum . . . 49

List of Algorithms

3.1 General Branch-and-Bound Framework Scheme 16
3.2 Processing Multiple Sheets with Branch-and-Bound Algorithms 17
3.3 Simple Algorithm Branching 18
3.4 CSA Algorithm Branching 21
3.5 Advanced Duplicate Detection 26



CHAPTER 1

Introduction

In industry it is often required to cut numerous elements out of a raw material sheet, usually made out of metal, glass, wood or similar. We describe an approach to find an optimal cutting pattern for a single sheet, subdividing it to obtain rectangular elements. These cuts made are restricted to guillotine cuts, which are explained in 1.2. The approach utilizes Branch-and-Bound and can be expanded to process multiple sheets and fulfil complex constraints.

This document is structured as follows: The next section describes the importance of the problem to solve as well as the goals to achieve. In Chapter 1.3 and following we describe the K-staged two-dimensional cutting stock problem with variable sheet size in detail and establish the naming conventions and structures used throughout this document. The literature survey in Chapter 2 displays previous work and related approaches. Chapter 3 contains the implementation of the Branch-and-Bound algorithm. Chapter 4 contains the achieved results, comparing the impact of the different measures taken. Chapter 5 finishes this thesis by summarizing the achieved work.

1.1 Aim of This Work

The aim of this work is to develop a *bottom-up* Branch-and-Bound approach to solve the K-staged two-dimensional cutting stock problem with variable sheet size (K-2CSV). It is a sub-type of the general two-dimensional cutting stock problem expanded by additional constraints. For further information see 1.3 below.

This variant of the problem is applicable to a wide range of real life situations in different industries like metal, paper or glass manufacturing. A typical application is cutting many units out of a material sheet in a way to minimize the material waste produced.

The algorithms discussed in this thesis produce an optimal result for a single sheet. The complete algorithm implemented in the K-2CSV framework of the *Algorithms and*

Complexity Group[1] solves as many sheets as are required for cutting out all elements, but does not guarantee a optimal solution over multiple sheets.

In real life applications it often is the case that different sheets with different costs are available, or it is necessary that remnants of previous assignments shall be used. Since Branch-and-Bound is an exact approach it allows to maximize the gain out of those sheets. Another common scenario is the need for very high quantities of small units, cut from a single sheet size. Again a Branch-and-Bound approach is advantageous over a heuristic approach since it minimizes the waste, which can add up quickly for a large number of sheets to cut.

As noted before this Branch-and-Bound approach is not suited to solve a K-2CSV problem over multiple sheets optimally. However it can help to find a better overall solution in combination with other approaches. The algorithm can be used to obtain an optimal solution for the first sheet which, in real world applications with high unit counts, will be needed many times. This means that the single sheet can be applied many times and a heuristic approach can be used to obtain a possibly better overall solution for the remaining units, while still minimizing waste.

An additional advantage of Branch-and-Bound is that it can be used to create solutions from already pre-filled sheets since it explores possibilities from a beginning state (usually an empty sheet). This state can be given by manufacturing constraints, other approaches used as preprocessing steps. It is also possible to set a pattern from another problem as the beginning state, combining the two different problems to save material. The explanation of the cutting tree in 1.4 illustrates this well.

Another use is optimizing solutions or sub-problems in the context of local search methods. In particular it can be applied in very large neighbourhood structures (VLNS) based on the *ruin-and-recreate* principle [34]. This approach allows for specifically targeting weak parts of the solutions and reoptimizing them. The optimal nature of solutions generated with Branch-and-Bound allows to take a part of existing solutions, remove that part and improve it by generating a new solution for the bounding rectangle with the still available parts.

To summarize, the goal is to create a Branch-and-Bound approach for the K-2CSV problem using a bottom-up strategy to optimally fill a single sheet.

1.2 Guillotine Cuts

Guillotine cuts are a special kind of cut. They are defined as *a division that must take place by a series of straight lines that extend from one edge of a rectangle to an opposing edge, parallel to the other two edges* [19]. That means that a guillotine cut always has to extend the whole width/height of the rectangle that it divides into smaller rectangles. Figure 1.1 and 1.2 below illustrate the difference. Pattern A has two clearly defined cuts that span the whole rectangle they subdivide, while pattern B does not have that property and therefore does not contain any guillotine cuts.

[1] https://www.ac.tuwien.ac.at/

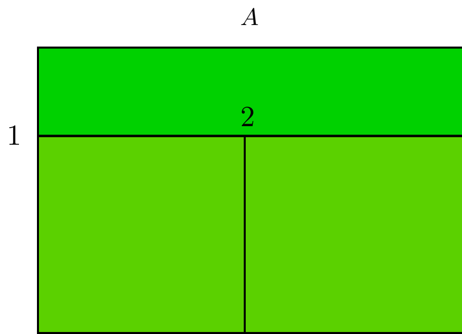


Figure 1.1: Guillotine Cuts

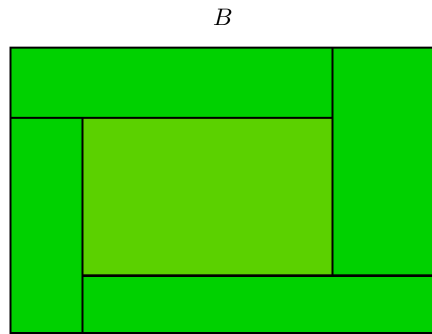


Figure 1.2: Non-Guillotine Cuts

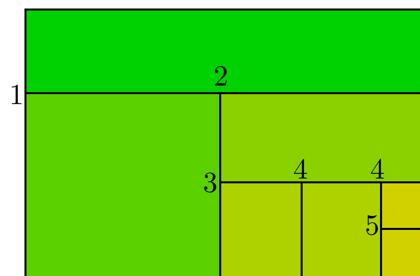


Figure 1.3: Guillotine Cut Stages

A pattern consisting only of guillotine cuts is easily executable on machinery and brings several advantages. It is more efficient since the machine only has to make cuts in one direction. Furthermore some machines are only able to perform such cuts. Some production techniques are only possible with guillotine cuts. An example of this is glass cutting, where the cuts get carved into the sheet and broken off afterwards. This is only possible along continuous cuts over the whole length since the glass would break on undesired locations otherwise.

A cutting pattern made out of guillotine cuts can be separated into *stages*. A cut at stage k divides a rectangle that was created at stage $k-1$. The first stage divides the whole sheet. Figure 1.3 illustrates this on a five-staged example.

1.3 Problem Definition

We are considering a *K-staged two-dimensional cutting stock problem with variable sheet size* (K-2CSV) in which we are given:

- a set of n_E rectangular *element types* $E = \{1, \ldots, n_E\}$, where each element type $E_i = (h_i, w_i, d_i, r_i) \in E$ is specified by a height $h_i \in \mathbb{N}^+$, a width $w_i \in \mathbb{N}^+$, a

demand $d_i \in \mathbb{N}^+$ and a flag $r_i \in \{0, 1\}$ indicating if elements of this type are rotatable by 90° ($r_i = 1$) or not ($r_i = 0$), i.e., width and height may be swapped;

- a set of n_T *stock sheet types* $T = \{1, \ldots, n_T\}$, where each sheet type $T_t = (H_t, W_t, q_t, R_t, c_t) \in T$ is specified by a height $H_t \in \mathbb{N}^+$, a width $W_t \in \mathbb{N}^+$, an available quantity $q_t \in \mathbb{N}^+$, a flag $R_t \in \{0, 1\}$ indicating if sheets of this type are rotatable by 90° ($R_t = 1$) or not ($R_t = 0$), i.e., width and height may be swapped, and a cost factor $c_t > 0$
- a parameter $K \in \mathbb{N}^+$ indicating the maximal number of stages of guillotine cuts for processing each sheet.

A feasible solution is a set of *(cutting) patterns* $P = \{P_1, \ldots, P_n\}$ describing an arrangement of all elements specified by E on a subset of the stock sheets specified by T without overlap and using guillotine cuts up to depth K only. Each pattern P_j, $j = 1, \ldots, n$, has an associated stock sheet type t_j and a quantity a_j specifying how often the pattern is to be applied, i.e., how many sheets of type t_j are cut following pattern P_j. More precisely, a cutting pattern is specified by a tree structure, which will be detailed in Section 1.4. The result of a singular application of the Branch-and-Bound algorithm is exactly one cutting pattern P_j.

We assume that at least one feasible solution exists, which implies in particular for each element type $i \in E$ that there is a stock sheet type $t \in T$ s.t. $0 < h_i \leq H_t$ and $0 < w_i \leq W_t$, or if $r_i = 1$ or $R_t = 1$ then $0 < h_i \leq W_t$ and $0 < w_i \leq H_t$ and the quantities of the stock sheet types are large enough.

For each sheet of type $t \in T$, we define the upper left corner to have coordinates $(0, 0)$ and the lower right corner with coordinates (H_t, W_t). Often the lower left corner is used as $(0, 0)$ in the literature, however this does not have any impact on the quality of the solutions. W.l.o.g., we assume that the direction of the guillotine cuts always is horizontal for odd stages (in particular the first stage) and vertical for even stages. Note that because of this it may be necessary to rotate the used stock sheet t to fulfill the K-stage requirement for the given instance. We refer to the rectangles resulting from stage-k cuts as *k-rectangles*. In particular, the rectangles resulting from stage-1 cuts are also called *strips*, the ones resulting from stage-2 cuts are called *stacks* and the requested target rectangles, resulting from stage-K cuts the latest, are called *elements*.

W.l.o.g., we assume that a cutting pattern only contains cuts necessary to cut out the requested elements defined by E, i.e., each cut goes along at least one element edge. Furthermore, we assume that in each (sub-)pattern possibly contained pure waste-rectangles are always located as far to the lower right as possible; in other words, all the requested elements are always shifted as far as possible towards the upper left corner.

For each stock sheet type $t \in T$, let $\sigma_t(P)$ be the number of used sheets of this type in P. Furthermore, for each cutting pattern P_j, for $j = 1, \ldots, n$, let ρ_j be the height of the possibly remaining waste-strip at the pattern's bottom ($\rho_j = 0$ if there is no such remainder).

The objective is to find a feasible set of cutting patterns P minimizing a cost function $c(P)$. In case of our basic problem variant, this cost function considers primarily the

numbers of used sheets weighted by the sheet type's cost factors and secondarily the largest remaining waste-strip of each sheet type:

$$\min\ c(P) = \sum_{t \in T} c_t \left(\sigma_t(P) - \frac{\text{maxremain}_t}{H_t} \right). \tag{1.1}$$

with

$$\text{maxremain}_t = \max\{0, \rho_j \mid j = 1, \dots, n \wedge t_j = t\}. \tag{1.2}$$

W.l.o.g., we do not need to explicitly consider blade width. It is sufficient to simply add the blade width to both h_i and w_i of each element type $i \in E$ and to both H_t and W_t of each sheet type $t \in T$ in a preprocessing step.

Similarly, trim cuts on the sheet edges can be taken care of during preprocessing and therefore do not need to be considered by the core algorithm: Subtracting the trim cut width twice from both H_t and W_t of each sheet type $t \in T$ is sufficient.

Weighted and Unweighted Problems

Problem instances for the K-2CSV problem can be either weighted or unweighted.

Each element type E_i has a profit value. For *weighted* problems, this value is given explicitly in the problems description. For *unweighted* problem this value is derived from the elements dimensions using its area.

We only consider unweighted problems in this thesis.

1.4 Cutting Tree

Each cutting pattern $P_j \in P$ is represented by a (cutting) tree structure (in the literature also referred to as *slicing tree*, see e.g. [18]). It consists of leaf nodes corresponding to individual elements (possibly in their rotated variants) and inner nodes being either *horizontal* or *vertical compounds* containing at least one subpattern. Vertical compounds always only appear at odd stages (levels), starting with stage one, and represent parts separated by horizontal cuts of the respective stage. Horizontal compounds always only appear at even stages and represent parts separated by vertical cuts. Each node thus corresponds to a rectangle of a certain size (h, w), which is, in case of compound nodes, the bounding box of the respectively aligned subpatterns. A pattern's root node always has a size that is not larger than the respective sheet size, i.e., $h \leq H_{t_j},\ w \leq W_{t_j}$. Sheets having identical cutting patterns are represented by the same cutting tree; remember that a_j represents the quantity of sheets cut according to pattern P_j. Similarly, compound nodes store congruent subpatterns only by one subtree and maintain an additional quantity.

Within its sheet, a pattern is always supposed to be aligned at the top left corner so that possible residual space appears to its right and bottom. In vertical compounds, specified substructures are always supposed to be arranged next to each other from top to bottom and aligned at their left edges, i.e., in case the substructures have different

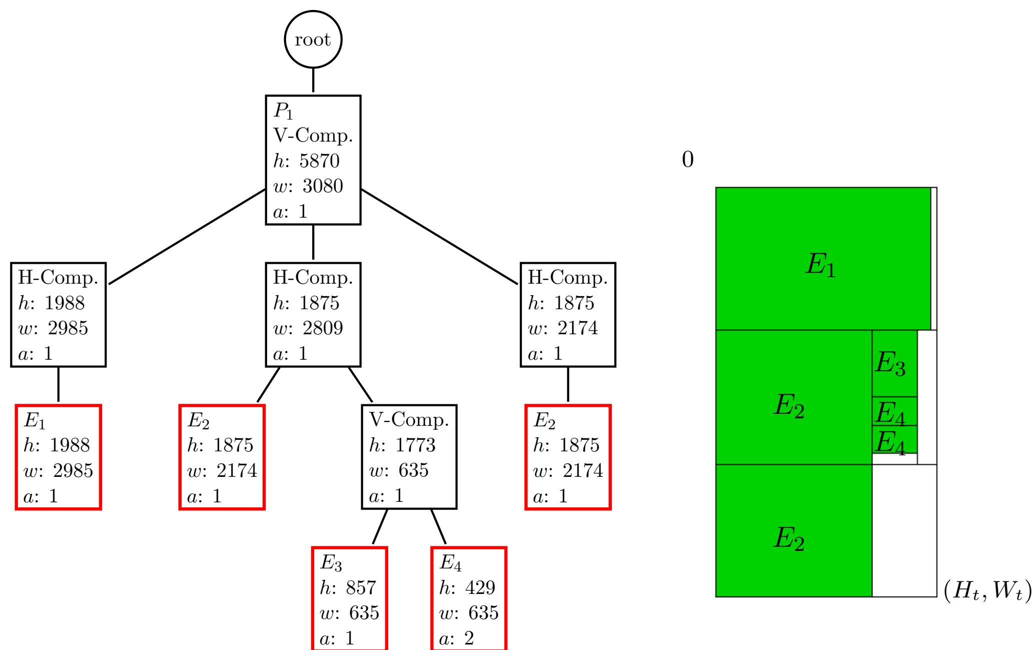


Figure 1.4: A three-staged cutting tree (left) and the corresponding single cutting pattern P_1 (right). The leaf nodes represent actual elements of types $E_1, \ldots, E_4$ obtained by the application of at most K stages of guillotine cuts. Note that the two elements of type E_4 are part of the same horizontal compound and need therefore only to be stored once in the tree. Figure adapted from [16]

.

widths, remaining space appears to their right. In horizontal compounds, substructures are always arranged from left to right and aligned at the top, i.e., in case the substructures have different heights, remaining space appears at the bottom.

Compound nodes with only one successor are required in cases where a cut is necessary to cut off a waste rectangle, otherwise they are avoided. Compound nodes are never directly followed by other compound nodes of the same type, i.e., horizontal or vertical, as they can be merged. Let us point out that in this tree structure, residual (waste) rectangles are never explicitly stored, but each compound node indirectly represents a possible additional residual rectangle when considering its embedding in the parent compound or sheet.

For representing a whole solution in an actual implementation the separate pattern trees can consistently be joined to a single tree by introducing an additional root node at tree level zero. This node represents a sheet compound whose subpatterns are the separate sheet patterns.

Figure 1.4 shows a three-staged cutting pattern for a single sheet and the cutting tree representing it.

1.4.1 Normal Form

We say a cutting pattern $P_j \in P$ is in *normal form*, if

(i) in each vertical compound the subpatterns are ordered by nonincreasing width and in case of equal width by nonincreasing height and

(ii) in each horizontal compound the subpatterns are ordered by nonincreasing height and in case of equal height by nonincreasing width.

Further ties in these orderings might be broken arbitrarily or in a way so that congruent subpatterns appear next to each other.

Considering the objective function as defined above, every feasible cutting pattern can be transformed into a pattern in normal form with identical objective value. It is therefore sufficient to only consider patterns in normal form in the optimization.

CHAPTER 2

Literature Survey

In [20] and [19] P.C. Gilmore and R.E. Gomory describe the general theory of two-dimensional cutting stock problems, including staged cutting and restrictions to guillotine cuts. The papers suggest different algorithms using a dynamic programming approach. In general column generation techniques are used.

Christofides and Whitlock describe a Branch-and-Bound approach in [4]. They present a tree-search algorithm for the constrained two-dimensional cutting stock problem, taking advantage of pattern symmetry and maximum amounts of pieces needed.

Building upon that Beasley [2] obtained improved bounds for the tree-search using a Lagrange relaxation of a zero-one integer programming formulation of the problem. Hifi and Zissimopoulos [25] create an improved tree-search algorithm by obtaining better bounds through dynamic programming procedures and reducing the problem to a one-dimensional knapsack. Hifi [23] and Cui et al. [8][9] improve upon exact algorithms and propose new variants. All of these algorithms are top-down, meaning they start with a sheet and successively cut it into smaller pieces, until the required elements are obtained.

Opposed to that we have the category of bottom-up approaches, discussed by Wang [37], Viswanathan and Bagchi [36], Cung et al. [12], Hifi [22][24] and Kang and Yoon [27]. These papers propose algorithms that combine pieces to fill the sheet, instead of dividing the sheet into smaller subsections to obtain the pieces.

In [12] Cung et al. elaborate on the top-down approach by Christofides and Whitlock [4] and more importantly the bottom-up approach by Viswanathan and Bagchi [36] and its improvement by Hifi [22]. The paper then describes a modified version of Hifis MVB (Modified Viswanathan Bagchi) algorithm. While the paper and its predecessors [36][22] do not concern themselves with the K-2CSV problem specifically, they still do offer a good basis to build a general Branch-and-Bound approach for the cutting stock problem and are often cited in the literature.

Kang and Yoon [27] describe an approach to obtain a better upper bound for cutting patterns. While most Branch-and-Bound algorithms utilize the relatively simple area-

based approach proposed by Hifi [22], Kang and Yoon respect the maximum obtainable height and respectively width in the leftover area not covered by the current pattern. The paper is based on the unconstrained cutting stock problem, however the methodology to calculate the bounds can be adapted for the K-2CSV problem.

A related problem is the *2-dimensional bin packing problem* (2BP) which packs element types into equally sized bins. There is no limit to the number of bins, and each element type has a fixed rotation. This problem category considers multiple bins, as opposed to the exact algorithms who fill exactly one bin/sheet.

In [28] Lodi et al. gather several approaches to solve the 2BP. The paper covers one-phase, two-phase and non-level heuristics as well as strip packing. Besides those the work also describes exact algorithms and metaheuristic techniques.

In another paper by Lodi et al. [29] they focus on the level-heuristics for the 2BP and the *2-dimensional strip packing problem* (2SP). Level based packing has the advantage of creating guillotine cuts, which relates directly to the cutting problem. Lodi et al. exploit the level restriction to obtain mathematical models for the level-restricted 2BP and 2SP that involves a polynomial number of variables. Through the use of those models they obtain a new lower bound through LP relaxation. They show that this bound dominates the standard area bound.

In [33] Puchinger et al. describe how to solve the *3-staged 2-dimensional Cutting Stock Problem* specialized in cutting glass. The restriction to three stages often occurs in real-world applications due to machine limitations. The algorithm creates solutions spanning multiple sheets. An additional constraint in this work is the partition of the element types into logical groups. Only elements from a maximum of three logical groups can be produced in an intertwined way, therefore the order in which the patterns are produced also matters. In the paper Puchinger et al. describe a greedy heuristic as well as two Branch-and-Bound algorithms and an evolutionary approach to solve the problem. Due to the complexity of the problem instances the Branch-and-Bound algorithms make use of heuristics, resulting in non-optimal solutions.

Puchinger and Raidl follow up on this paper in [32]. They describe integer linear programming formulations to solve both the restricted and the unrestricted *3-staged 2-dimensional Cutting Stock Problem*, without the additional constraints from the previous paper. Besides the polynomial-sized ILP models a column generation approach and a branch-and-price algorithm were developed.

In [5] Cintra et al. investigate several cutting stock problem variants, including 2CS and K-2CS. All variants use guillotine cuts and are considered with and without K-stage requirement and with and without rotation of elements. The Rectangular Knapsack (RK) problem is a variant where only one stock sheet is available and should be filled with element types. Cintra et al. propose two algorithms to solve the RK problem, of which only one is suited for the K-staged problem, but both utilizing *Discretization Points*. For the 2CS with and without variable sheet size and the 2-dimensional Strip Packing

problem a column generation approach is used. The focus in the paper lies on 2-, 3- and 4-staged patterns, however the proposed algorithms support K stages.

Wei et al. [38] explore a hybrid approach, combining a bottom-up approach, a top-down approach and a best-first heuristic into one algorithm. Other heuristic algorithms are described by Hong et al. [26], Cui et al. [11][10].

Recent studies have mostly dealt with heuristic approaches [18][26][10]. These algorithms use construction heuristics over an exact approach. While many of these findings are not directly applicable to a Branch-and-Bound algorithm they remain useful in hybrid combinations thereof.

Cui and Huang also explore T-shaped cutting patterns in [7]. They present an algorithm for the *Constrained 2-dimensional Cutting Problem*. There are four different versions of the algorithm presented, however only two are of practical relevance. The sheet is split into two segment: an X-segment, containing horizontal strips, and a Y-segment, containing vertical strips. The algorithm itself is split into two parts: a strip-generator that finds candidates for one of the segments, and a layout-generator to arrange the strips within the segment.

In [6] Cui develops a dynamic programming procedure to solve the *Unconstrained 3-staged 2-dimensional Cutting Problem* (UTDC), using guillotine cuts. The proposed algorithm solves three large knapsack problems, one for each cutting stage. At first the elements are packed into strips, then the strips to pack onto the sheet are considered from all generated strips. The chosen strips are then packed into the second cutting stage to obtain larger segments. Last those segments are packed onto the sheet.

Dolatabadi et al. [13] propose two exact algorithms to solve the *K-staged 2-dimensional Knapsack Problem* without element rotation, using guillotine cuts. It can also be classified as a *Guillotine 2-dimensional single large object placement problem*. As the name indicates the elements are fitted onto a single sheet. The algorithms are based on an exact recursive procedure that enumerates all packings of element-types on a stock sheet type. The patterns created *exact*, where all strips must have elements of equal height, as opposed to *non-exact* patterns where elements must trimmed using an additional cut.

The first algorithm stepwise converges an upper bound towards the optimal solution. The second algorithm utilizes branch-and-cut and is based on an ILP model of the problem.

CHAPTER 3

Solving Single Sheets by Branch-and-Bound

The algorithm was implemented in C++, extending an existing K-2CSV framework specialized on solving large scale problems on multiple sheets. The Branch-and-Bound approach will supplement the framework, since it does not contain any exact algorithms.

The K-2CSV framework is explained in the following chapter (3.1). Afterwards the general scheme of the Branch-and-Bound approach is explained in 3.2 and its subsections. A detailed explanation of the scheme can be found in algorithm 3.1. A simple enumerating Branch-and-Bound algorithm is explained in 3.3. The actual cutting stock algorithm follows in 3.4 and following.

3.1 Existing K-2CSV Framework

The base framework provides data structures and logic for the elements, patterns and in- and output of problem instances and solutions. Solutions can also be visualized via a SVG (Scalable Vector Graphics) image. Furthermore the framework contains several construction heuristics, based on the work by Charalambous and Fleszar [3] and Fleszar [18], and a Beam-Search approach.

The framework utilizes a scheduler to apply its different algorithms and supports methods to improve existing solutions or create new ones. This includes a ruin-and-recreate approach based on Dusberger [14]. The term ruin-and-recreate was coined by Schrimpf et al. [34] who applied this technique to solve the Travelling Salesman Problem (TSP) and the Vehicle Routing Problem (VRP). Exact algorithms like Branch-and-Bound supplement this kind of approach very well, since it guarantees an improvement of a recreated partial solution, if one exists, while being very fast on such a small scope.

3.2 Branch-and-Bound Framework

We build a framework for general Branch-and-Bound algorithms as the foundation of our work. The framework defines a general scheme for Branch-and-Bound algorithms, allowing the easy implementation of different approaches.
Due to the limitations imposed by the K-2CSV framework the Branch-and-Bound framework also provides functionality to solve a complete problem instance, filling multiple sheets. It contains the following components:

- The framework itself, providing the standard scheme and the integration into the K-2CSV framework
- The base class for Branch-and-Bound algorithms
- The node class representing the branches and the tree

3.2.1 Basis for Branch-and-Bound Algorithms

The base class provides functions for integration into the framework. These functions set up the algorithm and execute the different steps of the algorithm. Furthermore the base class also provides helper functions to ease the implementation of the concrete algorithm.

The interface functions entail:

- Setting up the algorithm
- Obtaining the next node to process
- Determining if a node should be pruned
- Branching on a node
- Calculating the score of a pattern
- Calculating the lower and upper bound of a pattern

The concrete algorithm implementations have to provide the logic behind these steps. For a detailed description on how those functions are called and their interplay with the rest of the framework see the next section (3.2.2).

Node class

The node is a simple structure to manage the branching tree. It only provides two functionalities: *adding* a node as child, which corresponds to creating a new branch, and *pruning* a child-node, removing it completely.

The nodes are used to manage the current state of the algorithm and as means of communication between the framework and the Branch-and-Bound algorithm.

3.2.2 Framework

The framework exposes two functionalities to the outside: filling a sheet and solving a complete problem instance.

The procedure for filling a sheet sets up a concrete Branch-and-Bound algorithm implementation and processes the sheet with the still available elements. Besides allowing the choice of different algorithms the framework also supports different parameters to modify the behaviour of said algorithms, as supported by the implementations. Additionally the framework features a *timeout* that allows to stop execution when a time-threshold is met. In that case the procedure returns with the currently best known result. *Usage of the timeout-feature removes the guarantee for an optimal solution!*

Additionally to the standard behaviour the framework also tracks several metrics of the algorithm.

The detailed process of filling a sheet can be found in algorithm 3.1. Note that line 18 removes all obsolete intermediate nodes whose children have been evaluated. It is guaranteed that only branches that are fully evaluated are removed by this, since only parent nodes without children (not counting the current node to be removed) are concerned.

Processing a complete problem instance

Since the K-2CSV Framework (see 3.1) requires us to solve a complete problem instance we propose a simple approach to apply the single-sheet Branch-and-Bound algorithm to a complete K-2CSV problem in algorithm 3.2.

Line 5 invokes the general Branch-and-Bound framework scheme previously described in algorithm 3.1. This function is invoked as often as it takes until all elements are assigned.

3.3 Simple Algorithm

We propose a simple enumerating algorithm to be used with the scheme 3.1. This algorithm provides a baseline for future results and the cutting stock algorithm in the next section (3.4). The bounds and traversal methods can be reused for any other algorithm, however they only serve as a starting point and equate to a near full enumeration of the search space.
This section is split into the different aspects needed by the algorithm. The *Branching* and the *Traversal* subsection explain the node expansion and traversal to process the problem.

The *Scoring* subsection elaborates on how the best solution is identified, while *Pruning & Bounds* deals with the basic methods in place to reduce the branching tree and therefore the search space.

Algorithm 3.1: General Branch-and-Bound Framework Scheme

Data: The algorithm A to use
The sheet T_T to fill
The remaining demands $D = d_1, ..., d_E$ of the element types $e_E \in E$
Result: The filled sheet

```
 1 initialize A with the sheet T_T and the demands D;
   /* The branching tree gets initialized by the algorithm   */
 2 root ⟵ root node of A's branching tree;
 3 node ⟵ root;
 4 last ⟵ null;
 5 best ⟵ 0;
 6 pattern ⟵ null;
 7 while true do
 8   if timeout occurred then
 9     break
10   end
     /* Evaluate node                                          */
11   if node has no children and node ≠ last then
12     score ⟵ A.evaluate(node);
13     if score > best then
14       best ⟵ score;
15       pattern ⟵ node;
16       A.updateLowerBound(pattern);
17     end
18     remove last evaluated node and all its parents without children;
19     last ⟵ node;
20   end
21   newNode ⟵ A.getNextNode(node);
     /* If the root node is returned it signals that the whole
        tree has been processed.  This condition can be
        replaced with any arbitrary termination condition.     */
22   if newNode = root then
23     break
24   end
25   if A.shouldPrune(newNode) then
26     prune newNode from the branching tree;
27     continue;
28   end
29   node ⟵ newNode;
30   A.expand(newNode);
31 end
32 set pattern as the pattern of the sheet T_T;
```

Algorithm 3.2: Processing Multiple Sheets with Branch-and-Bound Algorithms

```
   Data: Available Sheets T
   Element types E
   Result: A feasible Solution P
1  P ⟵ ∅;
2  while not all demands d_i of E fulfilled do
3  |  chose T_t ∈ T, q_t > 0;
4  |  if T_t = null then Error: Not enough sheets;
   |  /* Invoke the BnB Scheme:  Take a specific stock sheet
   |     type and a set of elements and fill the sheet with it.
   |     */
5  |  filledSheet ⟵fillSheet(T_t, E);
6  |  while element demands ≥ elements used in filledSheet and q_t > 0 do
7  |  |  subtract elements in filledSheet from demands;
8  |  |  reduce q_t by one;
9  |  |  P ⟵ P ∪ filledSheet;
10 |  end
11 end
12 return P;
```

3.3.1 Branching

This algorithm uses a *bottom-up* approach, meaning it obtains new patterns by combining existing ones, instead of subdividing the area. Each new branch is created by taking the pattern of the current node and combining it vertically or horizontally with another pattern. For this simple algorithm the strategy is to take the nodes pattern and combine it with each element type e_i horizontally and vertically. Since the cutting tree requires alternating horizontal and vertical combinations, the combination of a previous pattern with the new element in the same manner results in the extension of the pattern by the new element.

Furthermore the last child of the root of the cutting tree is extended with the new element. Due to the way patterns are constructed the last child node always contains the element added in the previous node. This ensures that the whole search space is covered.

Only element types which still have a residual demand left considering the node's pattern are taken into account. Residual demands are calculated at the beginning of the algorithm and represent the element demands that have not been fulfilled in the current solution so far. The complete procedure is detailed in 3.3.

3.3.2 Traversal

We use a depth-first traversal for this algorithm. Later algorithms (3.4) also support best-first traversal. If the last visited node has children, possibly added through the

Algorithm 3.3: Simple Algorithm Branching

```
   Data: Node n to expand
   Element Types E
   Residual Element Demands d_{R_i} ∈ D_R ⊆ D for the current sheet
 1 foreach e_i ∈ E do
 2 |   if d_{R_i} is fulfilled in n then continue;
 3 |   p ⟵ n.pattern;
 4 |   if p is empty then
   |   |   /* Root node has no pattern to combine, add the
   |   |      element as a pattern                          */
 5 |   |   n.addBranch(e_i);
 6 |   else if p is a vertical compound then
 7 |   |   n.addBranch(composeHorizontal(p, e_i));
 8 |   |   n.addBranch(extendCompound(p, e_i));
 9 |   |   p' ⟵ extend last subpattern in the first stage of p with e_i;
10 |   |   n.addBranch(p');
11 |   else if p is a horizontal compound then
12 |   |   n.addBranch(composeVertical(p, e_i));
13 |   |   n.addBranch(extendCompound(p, e_i));
14 |   |   p' ⟵ extend last subpattern in the first stage of p with e_i;
15 |   |   n.addBranch(p');
16 |   else if p is an element then
17 |   |   n.addBranch(composeVertical(p, e_i));
18 |   |   n.addBranch(composeHorizontal(p, e_i));
19 |   end
20 end
```

last branching call, we continue to its first child. If the last node has no children we recursively return to the next child in the parent above.

3.3.3 Scoring

The score, or fitness, of a pattern is determined by how well it utilizes the available space. A pattern is an element or a rectangular composition containing other patterns. As this thesis deals with unweighted problem instances only, the value of an element type e_i is defined by its area $w_i \cdot h_i$. The score of a pattern can easily be calculated by utilizing its *Cutting Tree representation*(1.4). Each node in the tree has an assigned width and height that it takes up on the sheet. The area that is *not* utilized can be calculated by taking the difference between the area of a node and the area of its children. By saving that data during tree construction the score of a pattern can be obtained in $O(1)$ by querying the root of the cutting tree.

We formally define $a(P_j)$ as the effective area used by the pattern P_j. Furthermore we define the function $in_P(P_j, e_i)$ which returns how often the element type e_i is present in the pattern P_j. This number represents the actual elements created by the pattern, not the number of nodes containing e_i in the cutting tree.

$$a(P_j) = \sum_{e_i \in E} h_i \cdot w_i \cdot in_P(P_j, e_i) \tag{3.1}$$

$$in_P(P_j, e_i) = \text{number of occurences of } e_i \text{ in } P_j \tag{3.2}$$

3.3.4 Pruning & Bounds

We define the following conditions. A node should be pruned if its pattern P_j...

- width w_i is larger than the sheet's width W_T
- height h_i is larger than the sheet's height H_T
- uses more than K cutting stages
- contains more elements of type e_i than its demand d_i
- has an upper bound lower than the current lower bound

The constraint for element demands can be resolved implicitly, since the branching strategy detailed in algorithm 3.3 does not create any branches for element types for which there are no more elements available.

The lower bound b_{lower} is updated whenever a new best pattern is found. It is obtained by using the score-function detailed in Equation3.1 in Section 3.3.3.

The upper bound b_{upper} is calculated by estimating the area that could potentially be used, if the remaining space would be filled perfectly. This is a very simple upper bound.

$$b_{lower}(P_j) = a(P_j) \tag{3.3}$$

$$b_{upper}(P_j) = a(P_j) + (H_t \cdot W_t - w_{max}(P_j) \cdot h_{max}(P_j)) \tag{3.4}$$

where $w_{max}(P_j)$ and $h_{max}(P_j)$ are the total width and height used by the normalized pattern P_j.

3.4 CSA Algorithm

We propose a *Cutting Stock Algorithm* (CSA) based on the literature. The algorithm builds upon the *Modified Viswanathan and Bagchi* (MVB) [22] algorithm version by Cung et al. [12].

As any bottom up approach, the MVB algorithm obtains new patterns by combining two previous patterns horizontally and vertically. To do so it holds a list named *clist*

that holds all previously constructed patterns. This includes patterns only containing a single element $e_i \in E$.

In the initialization step we add a branch for each element $e_i \in E$ to the root node. The elements will be added to the *clist* as they are encountered during traversal.

The pruning behaviour and constraints are taken from the Simple algorithm 3.3.

3.4.1 Branching

Each new branch is created by combining the current pattern to process, represented by the current node, with the other patterns from the *clist*. Whenever a new branch is explored (meaning the node does not violate any constraints) we add its pattern to the *clist* before branching. Therefore each pattern is also combined with itself. Each two patterns are combined once horizontally and once vertically. It is assumed that branching is called for valid patterns, that do not violate any constraints or bounds, only. This saves additional checks and keeps the *clist* size as small as possible. For performance reasons it makes sense to only add newly created patterns that do not violate the basic constraints like sheet-dimensions or element demands.

It is important to note that the *compose* functions in algorithm 3.4 also have to take care of correctly combining the two patterns. When combining two patterns of different height horizontally the resulting pattern requires an additional horizontal cut to cut out the smaller one. The same goes for vertical combinations and different widths respectively.

A basic duplication check is implemented into the Branch-and-Bound framework. It is provided by the K-2CSV framework and considers patterns with identical internal structure. Duplicate patterns like this are not added.

Whenever a new lower bound is discovered each node whose upper bound is below the new lower bound are removed, since they cannot result in a better pattern than the current best one.

3.4.2 Traversal

The CSA algorithm utilizes best-first traversal. To do so it keeps a sorted queue containing all leaf nodes.

Whenever a node is added during branching, we insert the node into the sorted queue. When the Branch-and-Bound framework requests the next node in algorithm 3.1 line 21 we pop the head of the queue and return it.

The sorting criteria is the nonincreasing score of the corresponding nodes pattern.

Since the returned node will be evaluated next, and since nodes are only added during branching and initialization it is ensured that the queue always contains *all* leaf nodes as well as that it *only* contains leaf nodes.

Algorithm 3.4: CSA Algorithm Branching

```
Data: Node n to expand
List clist of previous patterns
1 p ⟵ n.pattern;
2 clist.add(p);
3 foreach p_i ∈ clist do
4     p' ⟵ composeHorizontal(p, p_i);
5     if p' violates no constraints then n.addBranch(p') ;
6     p'' ⟵ composeVertical(p, p_i);
7     if p'' violates no constraints then n.addBranch(p'') ;
8 end
```

3.4.3 Demand-Aware Area Upper Bound

We develop a modified version of the upper bound proposed by Hifi [22]. The original bound was a dynamic programming procedure to solve a bounded knapsack problem. The bound is restricted to sheet-dimensions, but does not account for element type demands. The number of occurrences of an element type is constrained by how often it fits into the sheet vertically/horizontally.

Our modified approach takes the *remaining* area of the sheet, that is not occupied by its current pattern P_j, and calculates the best possible use of the area while taking element demands into account.

As we only deal with unweighted instances the profit of an element type e_i is defined by its area $w_i \cdot h_i$. This simplification allows us to take the combination of elements that requires the largest area, but still uses at most as much area as the sheet. Otherwise the upper bound can undershoot the optimal value, resulting in incorrect output.

$$
\begin{aligned}
b_{upper} = \quad & min & & \left(a(P_j) + \sum_{e_i \in E} w_i \cdot h_i \cdot n_i \right) \\
& \text{subject to} & & b_{upper} \leq W_t \cdot H_t \\
& & & 0 \leq n_i \leq (d_{R_i} - in_P(P_j, e_i)) \forall n_i
\end{aligned}
\tag{3.5}
$$

P_j is the pattern already present on the sheet and W_t and H_t are the sheet dimensions. w_i and h_i are the dimensions of the element $e_i \in E$. n_i is a value between zero and the maximum number the element type e_i can be used without violating demands considering the already present pattern P_j and possible other patterns in the solution. The goal is to minimize the resulting value by picking n_i for each e_i accordingly, while not packing more element types onto the sheet than there is space available.

n_i's are constructed by considering the remaining free space on the sheet and adding the element with the largest possible area not exceeding the space. The remaining free space after adding an element must not become negative. After the element has been

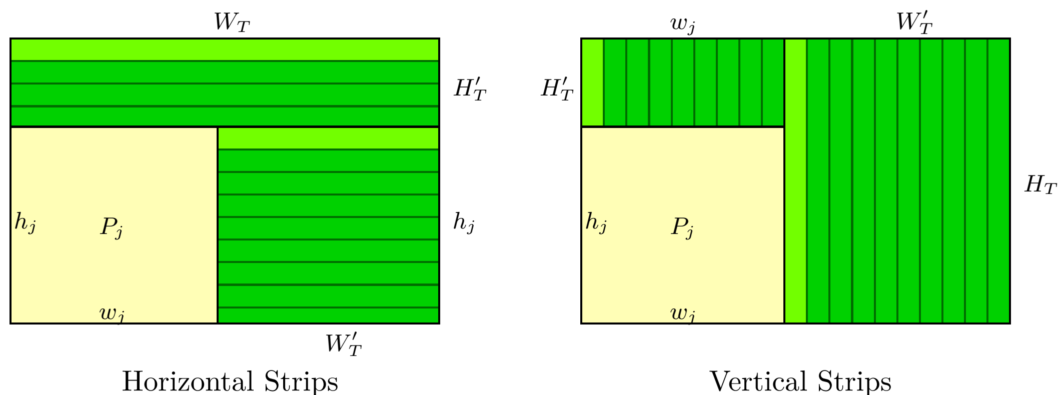


Figure 3.1: Strip Partitioning of Unused Pattern Space

added we subtract its area from the remaining free space. This is repeated until no element can be added under the constraints given in 3.5.

3.5 Algorithm Improvements

We propose several improvements to the CSA algorithm. This entails the lower bound, the upper bound and advanced detection of unnecessary patterns.

3.5.1 Initialize Lower Bound

As a first step we initialize the lower bound with a value greater than 0. This is done by obtaining a valid solution through a construction heuristic. Hifi [22] has already used such an approach, however for our approach we use an already existing construction heuristic. The K-2CSV framework provides an insertion heuristic [15] based on Fleszars work [18] which provides a simple lower bound for a very low computing time.

3.5.2 A Better Upper Bound Based on Element Combinations

We adapt a better upper bound by Kang & Yoon (2011) [27]. They propose a bound that not only takes the elements into account, but also the horizontal/vertical combination of those elements. To do so the area not covered by the pattern on the sheet is split into strips. Next the combination of elements that use the most of one strip is calculated and is then expanded to the available strips of the same size. These combinations only take one dimension into account, since we only care about the horizontal/vertical combination thereof. This approach is designed for the unconstrained variant of the problem and therefore does not take element demands into account.

Figure 3.1 shows the horizontal and vertical partitioning of the area into strips. We will detail the algorithm for the horizontal partitioning here. The vertical partitioning follows the same procedure, but with width and height swapped respectively.

First the area is split into the two rectangles that shape it: One spanning the whole width W_T and the unused height $H'_T = H_T - h_j$ of the pattern P_j. The other one using up the space next to the pattern, defined through the unused width $W'_T = W_T - w_j$ and the patterns height h_j. Those areas are then partitioned into strips of height 1 and width W_T and W'_T respectively.

The next step is to find the combination of elements that yield the highest profit when relativised to one strip. To obtain the relativised profit we divide the profit of the element type through its height (width for a vertical strip). Since we use unweighted elements we can simplify this to the width of the corresponding element. The profit of an arrangement of elements in a strip is therefore equal to the width of those elements. We find the widest arrangement w_{max} that fits W_T as well as the widest arrangement that fits W'_T (Equation3.7) and then apply their profit to each corresponding strip. The sum of those is the upper bound (Equation3.9)

The final upper bound is then the result of either the horizontal or the vertical process, whichever is lower.

As this bound does not take element demands into account it is possible to calculate the different arrangements (linear combinations) of elements and their lengths beforehand. This reduces Equation3.7 to a search or lookup in the cached lengths at runtime.

$$\begin{aligned} W'_T &= W_T - w_j \\ H'_T &= H_T - h_j \end{aligned} \qquad \text{where} \qquad \begin{aligned} w_j &= \text{width of } P_j \\ h_j &= \text{height of } P_j \end{aligned} \tag{3.6}$$

$$w_{max}(w) = max \left\{ \sum_{e_i \in E'} w_i : \sum_{e_i \in E'} w_i \leq w, E' \subseteq E \right\} \tag{3.7}$$

$$h_{max}(h) = max \left\{ \sum_{e_i \in E'} h_i : \sum_{e_i \in E'} h_i \leq h, E' \subseteq E \right\} \tag{3.8}$$

$$b'_{upper}(P_j) = H'_T \cdot w_{max}(W_T) + h_j \cdot w_{max}(W'_T) \tag{3.9}$$

$$b''_{upper}(P_j) = W'_T \cdot h_{max}(H_T) + w_j \cdot h_{max}(H'_T) \tag{3.10}$$

$$b_{upper}(P_j) = min\left(b'_{upper}(P_j), b''_{upper}(P_j)\right) \tag{3.11}$$

This bound can be further improved by also calculating the best possible usage of the height.

Usually not all strips of one width can be used, since often no combination of elements results in exactly the available height. We can account for this by calculating the best case usage of H'_T and using the leftover height as additional strips of width W'_T.

$$
\begin{aligned}
H''_T &= h_{max}(H'_T) \\
h'_j &= H_T - H''_T \\
b'_{upper}(P_j) &= H''_T \cdot w_{max}(W_T) + h'_j \cdot w_{max}(W'_T)
\end{aligned} \tag{3.12}
$$

The same is applied to the vertical partitioning, but with the leftover width.

3.5.3 Dominated Pattern

Kang and Yoon [27] and Young-gun et al. [39] propose a method of detecting *dominated patterns* which can be pruned to reduce the search space. A pattern P is dominated if P can always safely be replaced by the dominating pattern *P'* without changing the final soluion. The original constraints are:

$$w_P \geq w_{P'} \quad \textbf{and} \quad h_P \geq h_{P'} \tag{3.13}$$

$$profit(P) \leq profit(P') \tag{3.14}$$

However their work deals with unconstrained and both weighted and unweighted problems. We have to tighten the domination constraints for them to be applicable to constrained problems.

Since our elements are unweighted we can substitute the profit of a pattern with the effective area used *a(P)*. Afterwards we tighten the profit constraint 3.14 to:

$$a(P) < a(P') \tag{3.15}$$

It is important that patterns with equal area are not removed, since the element demands do not guarantee that equal area also means that they share the same optimal pattern for the remaining space. Without the relaxation a parent-pattern of the optimal solution might be removed.

The second constraint is tightened to:

$$w_P > w_{P'} \quad \textbf{and} \quad h_P > h_{P'} \tag{3.16}$$

This is for the same reason as above. The basic premise is to remove any patterns that are bigger than others while not providing any extra gain. However equal patterns cannot be removed because of the element demands.

Before we add a pattern P during branching we check if any pattern *P'* in *clist* dominates P. If both 3.15 and 3.16 hold we can remove P and do not add it.

3.5.4 Advanced Detection of Obsolete Patterns

We propose a method to recognize duplicate patterns to reduce the amount of branches created. We perform this check every time before adding a new node.

The check is based on the principle that we only combine existing patterns, but do not alter them. If two *normalized* (1.4.1) patterns share the same width, height and used space they are equal if they contain the same elements. This means that the two patterns represent the same pattern, but in a different arrangement. Since we only combine them with other patterns, it does not matter which of the two we use. For our K-2CSV problem we always keep the pattern that uses less stages, and discard the other pattern.

Algorithm 3.5 shows a possible implementation for the process.

3.6 Flexible Algorithm

The flexible algorithm serves as an intermediate layer between the base and the concrete algorithm implementation. It provides access to already implemented logic for the different algorithms and a way to access them in a parametrized manner. This allows for easy code reuse as well as the ability to perform multiple runs of an algorithm in different configurations without the need to change the source code.

Algorithm 3.5: Advanced Duplicate Detection

Data: Normalized pattern p to check
List *clist* containing normalized patterns
Result: Whether p is a duplicate of a pattern in *clist*

```
1  for ∀p′ ∈ clist do
       // Only consider patterns that share dimensions and area
2      if p′.width ≠ p.width or p′.height ≠ p.height or p′.waste ≠ p.waste then
3          continue;
4      end
5      duplicate ⟵ true;
6      for ∀e ∈ p do
7          if e ∉ p′ then
8              duplicate ⟵ false;
9              break;
10         end
11     end
       /* If all elements were in both patterns we found a
          duplicate                                          */
12     if duplicate then
           /* We replace the pattern in the clist if it has less
              stages                                         */
13         if p.stages < p′.stages then
14             clist.remove(p');
15             clist.add(p);
16         end
17         return p is a duplicate;
18     end
19 end
20 return p is not a duplicate
```

CHAPTER 4

Computational Results

The test runs were done on a cluster consisting of multiple nodes with 2x Intel Xeon E5540 (2.53 GHz Quad Core) and 24GB RAM each.

Each instance is processed on the same hardware. Each run has a timeout of 60 minutes, after which the current best known solution is accepted as the result.

For each instance it is recorded whether it can be solved within the timeout in any configuration and if the result is the optimal result. Since the algorithm is optimal, all runs that complete before the timeout must also have the optimal score.

The number of stages allowed, K, is set to nine. This value was chosen since it is the highest supported value and has the highest probability of discovering the optimal cutting pattern for the given element types. Therefore a high K value allows for the best results score-wise.

The algorithm is executed in the following configurations:

- Simple algorithm with *naive* bounds as described
- Simple algorithm with the *area* upper-bound (3.4.3)
- Simple algorithm with the *strip* upper-bound (3.5.2)
- CSA Algorithm with best first traversal (*BFS*), as described in 3.4
- CSA Algorithm with BFS and *strip* upper-bound (3.5.2)
- CSA Algorithm with BFS, strip upper-bound and no extra-check for constraint violations before adding branches (*check*). That means all branches are getting added, even though they will be pruned once they are considered.
- CSA Algorithm with depth first traversal (*DFS*) and strip upper-bound
- CSA Algorithm with BFS, strip upper-bound and domination-check (*dom.*, 3.5.3)

- CSA Algorithm with BFS and advanced duplication detection (*dupe*, 3.5.4)
- CSA Algorithm with BFS, strip upper-bound and all improvements

The following metrics are captured:

- Best Score
- If the instance can be completed within the timeout
- Stages required for the score
- Algorithm runtime
- Runtime until the best score is achieved
- Branches explored
- Leaf nodes evaluated

4.1 Instances

We use a total of 55 test instances taken from the literature to obtain comparable results. All Instances are constrained, unweighted instances and can be found at `ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/2Dcutting/2Dcutting.html`.

We use 38 instances from:

- 4 weighted instances from Hifi [22] that were modified to be unweighted by Cung et al. [12]:
  2s, 3s, A1s, A2s
- 3 unweighted instances from Hifi [22]:
  A3 - A5
- 2 weighted instances from Tschöke and Holthöfer [35] that were modified to be unweighted by Cung et al. [12]:
  STS2s, STS4s
- 7 instances from Cung et al. [12]:
  CHL1s - CHL4s, CHL5 - CHL7
- 11 instances from Fayard et al. [17]:
  CU1 - CU11
- 8 instances from Alvarez-Valdés [1]:
  ATP37, ATP39, Hchl3s - Hchl8s

- 1 instances by Wang [37]:
  W

- 2 instances from Oliveira and Ferreira [31]:
  OF1, OF2

Furthermore we take 17 unconstrained instances from the literature and constrain them to 50 pieces per element. This allows the algorithm to solve them while still retaining high possibility to obtain the optimal result as if the instance would be solved unconstrained. Those instances are denoted with an '*s*' suffix, just like the modified instances above.

The converted instances are:

- 1 from Herz [21]:
  Hs

- 1 from Hifi and Zissimopoulos [25]
  HZ1s

- 5 from Martello and Toth [30]:
  M1s - M5s

- 10 from Fayard et al. [17]:
  UU1s - UU10s

4.2 Results

Table 4.1 shows the optimal score as well as the best achieved score for each instance and its required stages. For instances that did not finish before the timeout the results of the highest score are taken. Furthermore the fastest achieved runtime for each instance is shown. The values for the optima are taken from Cung et al. [12], Alvarez-Valdés [1] and Kang and Yoon [27].

Changes made to detect duplicate patterns were implemented into the Branch-and-Bound framework during the development of the CSA algorithm. The utilities provided by the K-2CSV framework for this purpose also cache and simplify patterns. The Simple algorithm relies on the order in which the patterns are constructed, which is destroyed by this caching. Patterns which are needed to obtain the optimal result might therefore be changed or discarded by the duplicate detection. This can lead to a false, suboptimal solution as well as shortened runtimes for more complex patterns. The duplicate check cannot be separated from the framework easily. Furthermore the runtimes without any duplication check would be too long to be feasible for actual use on a more complex problem instance. We therefore do not consider the results of the Simple algorithm, but list them for completeness.

Table 4.1: Score, Stage and Timeout Results for $K=9$

Instance	Optimum	Best Score	Stages	Best Runtime [s]
2s	2.778	2.778	6	0,441
3s	2.721	2.721	4	0,322
A1s	2.950	2.950	2	0,026
A2s	3.535	3.535	6	0,127
A3	5.451	5.451	6	14,697
A4	6.179	6.179	7	52,871
A5	12.985	12.985	6	79,253
ATP37	387.276	387.276	6	3096,560
ATP39	268.750	268.750	7	3142,320
CHL1s	13.099	13.099	5	50,567
CHL2s	3.279	3.279	3	0,312
CHL3s	7.402	3.967	3	Timeout
CHL4s	13.932	8.801	2	Timeout
CHL5	390	390	4	0,205
CHL6	16.869	16.869	4	156,005
CHL7	16.881	16.881	5	509,392
CU1	12.330	12.330	4	0,175
CU2	26.100	26.100	2	0,571
CU3	16.723	16.723	4	1,248
CU4	99.495	99.495	6	2,688
CU5	173.364	173.364	4	1,252
CU6	158.572	158.572	2	0,130
CU7	247.150	247.150	4	0,518
CU8	433.331	433.331	6	0,358
CU9	657.055	657.055	4	0,081
CU10	773.772	773.772	6	25,443
CU11	924.696	924.696	3	701,695
Hchl3s	12.215	12.215	8	1082,300
Hchl4s	11.994	11.859	7	Timeout
Hchl5s	45.361	45.223	5	Timeout
Hchl6s	61.040	61.040	4	205,379
Hchl7s	63.112	63.112	6	617,082
Hchl8s	911	906	7	Timeout
Hs	12.348	12.348	4	0,349
HZ1s	5.226	5.226	2	1,136
M1s	15.024	15.024	2	0,070
M2s	73.176	73.176	6	0,356
M3s	142.817	142.817	4	0,064
M4s	265.768	265.768	2	0,099
M5s	577.882	577.882	2	0,090
OF1	2.737	2.737	6	0,064

Score, Stage and Timeout Results

Instance	Optimum	Best Score	Stages	Best Runtime [s]
OF2	2.690	2.690	4	0,298
STS2s	4.653	4.653	5	0,881
STS4s	9.770	9.770	6	25,597
UU1s	242.919	242.919	4	0,201
UU2s	595.288	595.288	4	0,046
UU3s	1.072.764	1.072.764	5	0,195
UU4s	1.179.050	1.179.050	6	1,580
UU5s	1.868.999	1.868.999	5	0,370
UU6s	2.950.760	2.950.760	3	0,096
UU7s	2.930.654	2.930.654	4	5,603
UU8s	3.959.352	3.959.352	5	0,849
UU9s	6.100.692	6.100.692	3	0,310
UU10s	11.955.852	11.955.852	4	5,439
W	2.721	2.721	5	0,328

As shown in Table 4.1, all instances besides *CHL3s*, *CHL4s*, *Hchl4s*, *Hchl5s* and *Hchl8s* could be solved optimally and, besides *CHL3s* and *CHL4s*, were close to the optimal value.

No instance required nine stages, however it is unknown how many stages are required for the optimal solution of the unsolved instances. Only five instances exceed six stages and more than half of the instances can be solved with four or less stages. We will further analyse this in below in Section 4.3.

Table 4.2: Runtime Results in Seconds

File	**Simple**			**CSA**						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
2s	7,38	31,34	2,50	-	1,10	1,07	1,10	3,42	0,51	0,44
3s	0,42	66,84	0,22	-	0,65	0,56	0,65	2,46	0,38	0,32
A1s	0,46	25,95	0,15	-	0,03	0,03	0,03	0,07	0,03	0,03
A2s	1,14	29,58	0,75	-	0,18	0,17	0,18	0,57	0,13	0,13
A3	3,46	378,98	3,28	-	42,36	43,20	42,13	211,74	16,04	14,70
A4	146,80	-	81,18	-	156,02	156,08	157,08	904,75	59,25	52,87
A5	136,97	-	45,27	-	297,03	281,18	298,81	-	90,76	79,25
ATP37	-	-	-	-	-	-	-	-	-	3096,56
ATP39	-	-	-	-	-	-	-	-	-	3142,32
CHL1s	390,48	-	913,05	-	226,19	232,02	228,05	-	64,73	50,57
CHL2s	0,08	1,71	0,04	-	0,66	0,66	0,66	2,08	0,34	0,31
CHL3s	-	-	-	-	-	-	-	-	-	-
CHL4s	-	-	-	-	-	-	-	-	-	-
CHL5	0,13	2,05	0,13	1199,86	0,94	0,91	0,94	3,25	0,20	0,21
CHL6	-	-	156,01	-	1760,03	1719,54	1923,30	-	276,34	210,04
CHL7	-	-	-	-	-	-	-	-	918,17	509,39
CU1	6,07	18,63	3,50	-	0,27	0,26	0,26	0,71	0,17	0,18

Runtime Results in Seconds

File	Simple			CSA						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
CU2	12,17	104,89	3,20	-	0,87	0,88	0,87	2,76	0,60	0,57
CU3	246,79	-	15,73	-	1,99	1,98	1,96	6,02	1,24	1,25
CU4	27,15	-	3,86	-	4,14	4,19	4,12	14,27	2,88	2,69
CU5	32,99	481,76	0,61	-	1,71	1,74	1,69	5,84	1,42	1,25
CU6	4,87	1019,44	0,88	-	0,15	0,15	0,15	0,39	0,13	0,13
CU7	1,83	94,66	0,70	-	0,87	0,87	0,86	3,14	0,67	0,52
CU8	1,39	-	0,51	-	0,50	0,51	0,50	1,67	0,41	0,36
CU9	7,90	22,35	0,71	-	0,09	0,09	0,09	0,26	0,08	0,08
CU10	1185,11	-	877,70	-	60,42	61,46	60,85	221,58	28,43	25,44
CU11	-	-	-	-	2325,43	2128,67	2284,43	-	895,65	701,70
Hchl3s	0,67	178,76	6,48	-	-	-	-	-	1137,97	1082,30
Hchl4s	149,21	-	138,58	-	-	-	-	-	-	-
Hchl5s	-	-	-	-	-	-	-	-	-	-
Hchl6s	-	-	624,83	-	929,97	981,61	982,73	-	262,73	205,38
Hchl7s	-	-	-	-	2220,31	2371,37	2279,67	-	800,44	617,08
Hchl8s	189,18	-	218,32	-	-	-	-	-	-	-
Hs	0,20	0,38	0,04	-	461,62	487,76	482,90	-	0,36	0,35
HZ1s	0,01	-	0,01	-	-	-	-	-	1,10	1,14
M1s	0,31	0,57	0,07	-	0,78	0,79	0,78	2,72	0,07	0,07
M2s	0,81	16,65	0,12	-	1,39	0,99	1,39	4,15	0,36	0,36
M3s	0,12	0,53	0,03	863,78	0,11	0,11	0,11	0,30	0,08	0,06
M4s	0,17	0,49	0,06	-	0,98	0,99	1,00	3,25	0,11	0,10
M5s	0,42	0,71	0,09	-	0,96	0,96	0,97	3,35	0,10	0,09
OF1	0,06	0,41	0,04	266,30	0,10	0,09	0,10	0,27	0,07	0,06
OF2	0,40	0,51	0,08	423,19	0,55	0,59	0,55	1,83	0,37	0,30
STS2s	0,69	28,24	0,46	-	1,35	1,50	1,35	4,75	1,02	0,88
STS4s	1507,97	-	-	-	157,32	136,91	155,37	-	66,35	25,60
UU1s	2,51	97,05	0,47	-	0,36	0,39	0,36	1,15	0,24	0,20
UU2s	5,35	43,84	5,75	-	0,06	0,06	0,06	0,14	0,05	0,05
UU3s	0,50	24,45	0,92	-	0,27	0,26	0,26	0,82	0,20	0,20
UU4s	4,44	360,42	18,01	-	2,77	2,83	2,75	9,56	1,87	1,58
UU5s	24,60	-	5,19	-	0,51	0,51	0,51	1,41	0,37	0,37
UU6s	13,36	222,47	0,91	-	0,11	0,11	0,11	0,30	0,11	0,10
UU7s	394,00	439,10	27,76	-	11,16	11,68	11,65	37,32	6,90	5,60
UU8s	126,04	-	72,42	-	1,41	1,53	1,40	4,41	0,89	0,85
UU9s	26,67	-	7,78	-	0,39	0,39	0,39	1,04	0,32	0,31
UU10s	173,49	-	18,57	-	12,78	12,87	12,83	40,81	6,44	5,44
W	0,34	2,08	0,11	459,10	0,70	0,72	0,70	2,72	0,38	0,33

Table 4.2 shows the runtime required to find the solution. Instances that could not finish within the timeout are denoted with '-'.

The Simple algorithm produces very fast results for some instances, however due to the implementation changes described above only three instances resulted in the correct result: *CHL3s*, *CHL4s* and *HZ1s*. Both *CHL3s* and *CHL4s* terminated by running into the timeout, surprisingly only the unreliable Simple algorithm managed to find the

optimal solution for those. Closer inspection of the created pattern shows that roughly half the sheet is covered by a large amount of small elements. This is an unfavourable problem instance for the CSA algorithm, since it gives the algorithm plenty of empty space to combine patterns, resulting in a much longer runtime required.

Overall the strip-bound provides the greatest improvement and is necessary to solve many of the instances. What is surprising, however, is that the area bound stays far behind the naive bound. This might be due to implementation details or caused by slow demand calculations. The strip bound represents an improvement in either case.

The results show that the Best-First-Search has next to no impact. DFS sometimes is a bit slower as well as being a bit faster sometimes, leading to no clear conclusion whether BFS provides any improvement.

The same outcome can be observed with respect to the domination-check, however the general tendency seems to be a minor improvement there.

The extra-check to prevent nodes from being added to the tree if they would have been pruned anyway shows a definite impact on runtime. This is expected, since it saves memory and sorting operations. The improvement it provides is much greater than expected. Most striking here is instance *STS4s* which could be solved in 157 seconds with just the strip bound, but can't be solved within the timeout with the extra-check disabled.

The advanced duplicate check also has a very positive impact. The runtime gain from the reduced search space is much greater than the additional calculations required for the duplicate check. Extreme cases like instance *Hs* and *W* show that the gains can be massive, but instances on which the algorithms generally run faster also show definite benefits from it.

The combined results of all improvements show that it is not solely the duplicate check that reduces runtime, but their combination. The other improvements clearly complement the duplicate check, allowing for a greater speedup. The instances *ATP37* and *ATP39* only become solvable within the timeout once all improvements are taken into account.

Overall the results confirm the expectations that the CSA Algorithm outperforms the Simple algorithm. Instances like *CHL3s* and *CHL4s* also show its worst case scenario: The algorithm does not handle instances with large sheets and many small element types that do not cover it completely well.

Table 4.3: Runtime Until Best Solution Found in Seconds

File	**Simple**			**CSA**						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
2s	2,53	30,77	2,46	1151,08	1,03	1,00	1,03	3,24	0,48	0,41
3s	0,36	11,50	0,13	2452,55	0,64	0,56	0,64	2,43	0,37	0,31
A1s	0,00	0,66	0,00	0,47	0,01	0,01	0,01	0,04	0,01	0,01
A2s	0,98	27,55	0,00	482,35	0,16	0,15	0,16	0,51	0,12	0,11
A3	0,00	2,31	0,00	1330,15	41,30	41,99	41,09	206,61	15,85	14,53
A4	0,01	1924,66	0,00	2325,96	150,88	150,90	151,94	878,68	56,99	50,71
A5	0,56	130,75	0,14	721,19	260,46	241,76	261,93	88,59	80,61	69,19

Runtime Until Best Solution Found in Seconds

File	Simple			CSA						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
ATP37	520,57	278,10	1125,48	83,49	2973,48	3018,97	3230,97	29,44	3458,99	3055,94
ATP39	0,27	57,68	2,10	242,05	3211,68	3034,12	3559,03	183,57	639,54	3042,36
CHL1s	13,18	467,58	0,22	2729,31	202,47	208,13	204,17	102,93	52,82	38,41
CHL2s	0,00	1,08	0,00	386,42	0,47	0,45	0,47	1,45	0,26	0,23
CHL3s	217,95	262,52	346,42	2342,96	2278,39	2555,57	2417,95	1,10	1919,28	1153,79
CHL4s	449,96	534,92	343,07	59,42	58,96	2743,45	59,43	167,13	30,89	1379,38
CHL5	0,08	0,86	0,08	337,60	0,40	0,37	0,40	1,37	0,13	0,14
CHL6	17,27	595,35	31,24	3011,65	1759,74	1719,25	99,98	262,81	276,12	209,82
CHL7	413,36	45,35	22,23	2527,96	3494,28	3599,70	3556,40	279,68	917,87	509,09
CU1	3,05	4,72	0,01	504,01	0,05	0,05	0,05	0,15	0,04	0,04
CU2	4,03	6,33	3,01	2830,73	0,83	0,83	0,82	2,63	0,59	0,56
CU3	6,38	1374,21	15,08	3235,34	1,83	1,83	1,80	5,59	1,20	1,21
CU4	0,00	1902,52	0,00	274,56	2,83	2,85	2,81	9,70	2,09	1,94
CU5	0,06	47,73	0,45	376,67	1,62	1,65	1,59	5,56	1,36	1,20
CU6	0,41	32,47	0,77	76,14	0,13	0,13	0,13	0,34	0,11	0,11
CU7	1,79	93,48	0,00	205,07	0,73	0,79	0,72	2,67	0,59	0,46
CU8	0,03	312,74	0,01	42,89	0,40	0,41	0,39	1,33	0,35	0,31
CU9	6,39	15,09	0,00	5,50	0,03	0,03	0,03	0,09	0,03	0,03
CU10	587,92	575,49	843,28	65,30	49,50	54,31	49,87	177,86	25,70	23,07
CU11	1841,50	1625,60	0,00	271,23	1565,96	1448,27	1529,76	22,62	642,84	480,58
Hchl3s	0,02	0,12	0,01	3064,20	1773,20	3038,01	1905,28	100,30	428,83	407,85
Hchl4s	0,11	1,89	75,14	97,06	379,75	349,96	385,43	181,27	1040,19	3133,74
Hchl5s	0,00	0,00	0,00	115,80	379,91	435,48	388,99	674,25	2843,14	2674,75
Hchl6s	5,40	30,49	0,01	84,56	915,75	971,56	967,41	751,33	256,61	199,84
Hchl7s	2417,56	0,41	1542,54	2238,55	2088,03	2244,02	2144,87	659,62	737,02	549,32
Hchl8s	161,09	383,00	43,52	1377,18	192,07	222,85	193,48	3118,78	2713,03	1930,89
Hs	0,01	0,01	0,00	541,01	2,02	2,64	2,01	6,82	0,35	0,34
HZ1s	0,00	0,00	0,00	0,08	0,01	0,01	0,02	0,04	0,01	0,01
M1s	0,00	0,49	0,00	0,01	0,00	0,00	0,00	0,00	0,00	0,00
M2s	0,00	1,45	0,00	47,35	1,38	0,93	1,38	4,10	0,36	0,35
M3s	0,04	0,19	0,00	23,71	0,07	0,08	0,07	0,20	0,05	0,04
M4s	0,00	0,41	0,00	0,16	0,01	0,01	0,01	0,01	0,00	0,00
M5s	0,14	0,63	0,08	0,15	0,00	0,01	0,00	0,01	0,00	0,00
OF1	0,04	0,38	0,02	144,67	0,08	0,08	0,08	0,24	0,06	0,06
OF2	0,30	0,45	0,03	173,57	0,52	0,57	0,53	1,76	0,35	0,28
STS2s	0,00	0,00	0,00	2041,10	1,25	1,40	1,26	4,47	0,94	0,80
STS4s	278,32	104,44	0,03	1799,72	156,96	136,55	155,01	1816,41	66,15	25,39
UU1s	1,53	21,46	0,43	1235,01	0,34	0,37	0,34	1,07	0,22	0,19
UU2s	2,53	3,71	0,00	824,75	0,02	0,02	0,02	0,05	0,02	0,02
UU3s	0,05	0,01	0,00	1721,96	0,10	0,09	0,10	0,29	0,09	0,08
UU4s	1,54	72,36	11,66	3330,60	2,47	2,55	2,45	8,53	1,75	1,46
UU5s	0,00	2988,44	0,00	1342,75	0,40	0,39	0,41	1,10	0,29	0,29
UU6s	8,83	45,32	0,02	711,79	0,06	0,06	0,06	0,17	0,06	0,05
UU7s	226,28	132,02	0,01	586,56	6,10	6,66	6,39	22,50	4,15	3,17
UU8s	118,35	1410,49	71,57	0,02	0,83	0,94	0,83	2,69	0,58	0,54
UU9s	0,00	92,73	0,00	3,46	0,06	0,06	0,06	0,19	0,06	0,05

Runtime Until Best Solution Found in Seconds

File	**Simple**			**CSA**						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
UU10s	0,01	7,78	0,01	125,15	5,75	6,07	5,76	18,31	3,74	3,41
W	0,20	0,96	0,03	290,85	0,69	0,72	0,69	2,69	0,37	0,32

Table 4.3 display the runtime until the best result is found. The results indicate that, overall, improved solutions are found throughout the whole process. Most instances take up to 70% - 99% of their runtime to find the optimal solution. Outliers which only required a fraction of their overall runtime to find the best result usually are far below 1 second runtime.

The most interesting instance in these results is *CU11*, which only took 68% (480 of 701 seconds) of its runtime to find the optimal solution. Generally the instances with runtimes above a few seconds also require most of the time to find the solution.

Another thing to note is the instances which could not be solved within the timeout. *CHL3s* and *CHL4s* both have not found an improved solution in the last 35 minutes of their runtime. This may indicate that the slowdown through the combinatorial explosion of patterns becomes too big to solve those instances.

Table 4.4: Branches Explored

File	**Simple**			**CSA**						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
2s	48.727	191.844	16.199	62.428	1.674	1.644	1.674	1.674	1.025	908
3s	1.022	40.443	622	49.405	865	824	865	865	640	570
A1s	1.199	17.605	438	50.010	163	166	163	163	153	152
A2s	1.866	18.325	1.349	51.044	479	465	479	479	407	383
A3	5.329	113.380	4.574	52.807	7.511	7.585	7.511	7.511	4.538	4.217
A4	103.659	593.196	64.778	51.401	14.469	14.480	14.469	14.469	8.928	8.208
A5	249.966	2.218.391	84.825	59.613	23.030	22.317	23.030	13.933	12.792	11.299
ATP37	837.091	489.008	1.117.401	54.458	57.482	55.394	53.840	13.578	93.663	85.279
ATP39	898.606	776.981	882.803	52.966	63.111	62.138	57.324	13.747	60.673	70.033
CHL1s	380.601	705.099	739.752	63.036	22.730	22.576	22.730	11.664	12.770	10.802
CHL2s	556	10.290	345	50.414	799	788	799	799	556	533
CHL3s	4.309.204	3.731.836	4.733.964	41.908	42.426	41.380	41.199	11.597	38.205	37.578
CHL4s	5.361.989	11.442.249	8.560.223	39.997	41.129	39.625	39.987	7.986	36.860	36.605
CHL5	1.128	16.298	1.042	31.385	1.198	1.155	1.198	1.198	515	515
CHL6	1.484.615	640.154	143.589	60.829	61.964	60.642	61.964	15.987	25.989	21.494
CHL7	1.236.560	512.886	617.004	68.392	75.778	71.238	74.013	17.345	48.582	34.436
CU1	6.268	15.051	4.592	52.421	488	486	488	488	389	386
CU2	12.852	66.540	4.274	51.280	1.122	1.129	1.122	1.122	868	822
CU3	122.554	408.984	14.734	51.431	1.522	1.536	1.522	1.522	1.138	1.125
CU4	19.121	403.131	3.691	51.666	2.244	2.275	2.244	2.244	1.850	1.734
CU5	26.444	135.391	716	52.052	1.529	1.566	1.529	1.529	1.362	1.224
CU6	4.461	201.092	1.040	53.976	404	412	404	404	368	358
CU7	2.503	49.406	1.321	52.296	955	955	955	955	820	693
CU8	2.107	189.067	835	51.459	831	836	831	831	724	654
CU9	7.791	15.659	1.404	51.262	327	324	327	327	300	281

Branches Explored

File	**Simple**			**CSA**						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
CU10	397.492	333.981	167.115	55.678	7.916	7.947	7.916	7.916	5.345	4.938
CU11	729.994	367.112	384.886	54.482	62.135	62.321	62.135	9.730	36.238	31.203
Hchl3s	4.356	667.781	30.430	69.127	66.586	62.079	64.542	12.222	31.294	29.182
Hchl4s	710.617	2.693.216	667.676	56.776	63.931	60.939	63.239	10.196	51.794	56.843
Hchl5s	212.242	323.290	675.706	56.294	71.453	72.387	69.437	16.154	76.141	76.557
Hchl6s	621.147	612.235	233.411	53.668	44.556	45.903	44.556	16.708	22.591	18.293
Hchl7s	965.496	270.853	705.853	54.402	88.316	90.981	88.316	14.512	50.228	38.429
Hchl8s	1.927.204	6.761.109	1.828.451	69.160	61.259	60.106	59.211	13.717	57.525	52.076
Hs	2.521	3.954	516	71.322	21.175	21.440	21.175	9.923	772	742
HZ1s	95	2.647.012	77	50.850	53.124	52.385	51.808	11.791	819	819
M1s	2.143	3.098	445	49.531	799	800	799	799	208	206
M2s	4.670	83.586	709	51.550	1.123	973	1.123	1.123	527	519
M3s	796	2.968	193	27.496	379	401	379	379	300	267
M4s	1.053	2.675	421	49.965	889	894	889	889	258	246
M5s	2.413	3.703	583	49.465	878	883	878	878	251	233
OF1	540	3.114	302	14.951	368	375	368	368	286	269
OF2	2.827	3.709	590	21.268	864	903	864	864	681	611
STS2s	1.320	30.409	851	54.818	1.423	1.510	1.423	1.423	1.194	1.083
STS4s	573.415	847.614	603.140	56.777	19.007	17.906	19.007	8.294	12.080	7.161
UU1s	3.455	39.330	824	50.647	643	661	643	643	507	451
UU2s	6.221	34.961	6.921	53.970	217	217	217	217	184	179
UU3s	1.004	17.127	1.621	52.623	567	564	567	567	481	438
UU4s	5.435	125.423	19.039	52.232	1.677	1.683	1.677	1.677	1.333	1.195
UU5s	18.138	319.601	4.828	51.681	609	613	609	609	504	496
UU6s	12.155	97.458	1.336	52.439	350	349	350	350	334	314
UU7s	145.046	192.507	20.392	52.155	3.488	3.591	3.488	3.488	2.669	2.314
UU8s	44.069	265.802	39.767	51.913	1.167	1.215	1.167	1.167	896	851
UU9s	15.640	306.372	5.929	52.907	598	597	598	598	546	512
UU10s	77.820	161.361	12.963	51.531	3.595	3.605	3.595	3.595	2.496	2.238
W	808	4.222	323	22.350	901	919	901	901	646	584

Table 4.4 shows the number of branches created during the process. Nodes whose addition was prevented by the extra-check are not counted.

A direct correlation between the runtime and the branches explored is visible. The most branches achieved within the timeout for any CSA configuration is roughly 70.000 branches, while the Simple algorithm, even with the reduced search space, can go well past millions.

This leads to the conclusion that the Simple algorithm can process the single branches much faster than the CSA algorithm with its additional checks. However it is apparent that the CSA algorithms require much fewer branches, which can be attributed to those checks, resulting in a greater speedup overall as seen before.

Table 4.5: Leaf Nodes Evaluated

File	Simple			CSA						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
2s	24.711	89.120	8.207	45.381	1.009	994	1.009	1.009	612	539
3s	619	23.668	364	43.323	669	640	669	669	484	428
A1s	711	11.127	278	46.010	121	124	121	121	114	113
A2s	1.278	12.423	903	48.094	340	330	340	340	288	267
A3	3.910	76.229	3.069	49.464	6.118	6.171	6.118	6.118	3.661	3.377
A4	64.265	366.330	39.591	46.955	11.265	11.281	11.265	11.265	6.893	6.299
A5	145.779	1.287.057	48.974	51.031	16.313	15.774	16.313	10.060	9.123	7.964
ATP37	412.999	222.338	525.678	47.770	51.378	49.387	48.066	11.522	73.074	64.414
ATP39	450.892	412.293	428.341	48.135	56.485	55.984	51.324	12.539	53.479	52.331
CHL1s	232.791	435.241	454.572	52.899	17.313	17.261	17.313	9.249	9.393	7.816
CHL2s	318	5.205	202	42.132	578	574	578	578	391	377
CHL3s	2.286.281	1.998.721	2.364.564	27.457	27.798	26.487	26.990	7.088	24.966	24.888
CHL4s	2.723.718	5.792.733	4.205.990	27.445	28.179	26.585	27.437	6.453	25.310	30.954
CHL5	505	7.688	465	20.869	701	674	701	701	312	312
CHL6	970.920	409.973	87.641	54.360	46.745	45.588	46.745	12.782	20.252	16.590
CHL7	867.632	357.525	393.584	60.404	67.409	63.498	65.931	15.172	40.584	28.037
CU1	4.355	10.568	3.219	47.547	343	343	343	343	269	266
CU2	9.088	46.019	3.029	47.863	799	812	799	799	616	577
CU3	89.564	279.959	9.309	48.524	1.033	1.043	1.033	1.033	783	772
CU4	13.915	296.272	2.539	48.661	1.697	1.722	1.697	1.697	1.387	1.292
CU5	19.148	93.919	518	49.563	1.164	1.197	1.164	1.164	1.030	907
CU6	3.608	149.640	715	51.375	293	301	293	293	267	257
CU7	1.700	33.455	891	48.134	685	685	685	685	584	490
CU8	1.474	119.489	581	48.235	634	637	634	634	561	497
CU9	4.926	10.774	951	47.041	232	230	232	232	211	194
CU10	257.685	184.011	98.833	49.342	6.124	6.141	6.124	6.124	4.091	3.755
CU11	502.730	260.071	234.982	49.951	49.388	49.595	49.388	7.777	28.288	24.266
Hchl3s	2.003	342.583	15.709	51.973	48.560	45.701	47.078	8.485	21.406	19.969
Hchl4s	356.516	1.374.143	330.872	40.150	48.101	46.082	47.442	6.955	38.151	41.316
Hchl5s	155.014	198.406	476.539	49.393	58.902	59.799	57.270	13.801	59.889	59.360
Hchl6s	440.650	404.700	156.076	45.636	35.201	36.291	35.201	13.812	17.374	13.877
Hchl7s	658.853	169.730	472.933	50.253	71.616	74.042	71.616	12.667	39.169	28.865
Hchl8s	921.549	3.290.748	875.997	49.378	41.780	43.204	40.428	8.969	32.586	32.291
Hs	950	1.509	209	56.418	16.085	15.698	16.085	7.729	465	445
HZ1s	15	950.111	14	45.586	46.727	46.513	45.528	10.404	529	529
M1s	1.061	1.479	237	45.264	602	591	602	602	132	131
M2s	2.147	41.229	327	44.458	755	653	755	755	333	327
M3s	445	1.609	116	24.381	269	279	269	269	204	177
M4s	521	1.315	222	45.316	676	669	676	676	168	158
M5s	1.183	1.808	288	44.548	666	658	666	666	164	151
OF1	254	1.505	140	10.072	228	228	228	228	179	164
OF2	1.508	2.015	323	16.853	619	639	619	619	485	431
STS2s	912	18.419	578	50.098	1.046	1.120	1.046	1.046	869	773
STS4s	376.832	519.520	371.876	54.003	16.215	15.184	16.215	7.610	10.085	5.688
UU1s	2.288	25.460	567	46.395	475	486	475	475	375	325

Leaf Nodes Evaluated

File	Simple			CSA						
	naive	*area*	*strip*	*BFS*	*strip*	*DFS*	*dom.*	*check*	*dupe*	*all*
UU2s	4.301	24.910	4.561	49.778	129	129	129	129	113	108
UU3s	739	11.807	1.056	47.910	410	406	410	410	342	302
UU4s	3.892	87.036	12.361	47.640	1.288	1.292	1.288	1.288	1.009	899
UU5s	13.537	231.650	3.389	48.510	398	400	398	398	335	327
UU6s	7.813	60.616	915	50.404	265	263	265	265	255	235
UU7s	104.025	133.771	13.963	48.513	2.663	2.742	2.663	2.663	2.046	1.755
UU8s	32.006	184.969	27.570	48.963	854	893	854	854	653	617
UU9s	11.767	205.641	4.148	48.781	443	441	443	443	404	372
UU10s	56.610	114.182	8.904	47.423	2.711	2.708	2.711	2.711	1.875	1.664
W	467	2.242	191	18.422	675	692	675	675	468	422

Table 4.5 shows how many of the explored branches were leaf nodes containing finished solutions. Again the results show that the CSA algorithm generally needs to consider fewer branches than the Simple algorithm.

4.3 Restricted Stages

We restrict the stages to a lower value to determine their impact on runtime and score.

We reduce the configurations used for this to the main four ones: Simple naive, Simple strip, CSA strip and CSA with all improvements.

For each instance we denote the deviation from the optimal score under the stage restriction as well as a comparison of the runtime to *K=9*.

4.3.1 6 Stages

Since most instances can be solved in six stages or lower we first restrict K to six and see if the runtime improves.

Table 4.6: Runtime Results for *K=6* in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	*K=6*	*K=9*	*K=6*	*K=9*	*K=6*	*K=9*	*K=6*
2s	-	7,38	7,41	2,50	2,58	1,10	0,52	0,44	0,26
3s	-	0,42	0,49	0,22	0,22	0,65	0,09	0,32	0,06
A1s	-	0,46	0,49	0,15	0,15	0,03	0,02	0,03	0,02
A2s	-	1,14	1,28	0,75	0,82	0,18	0,16	0,13	0,12
A3	-	3,46	3,85	3,28	3,43	42,36	42,19	14,70	14,77
A4	0,99	146,80	167,91	81,18	87,41	156,02	279,28	52,87	84,26
A5	-	136,97	145,96	45,27	46,15	297,03	275,23	79,25	73,45
ATP37	-	-	-	-	-	-	-	3096,56	2761,64
ATP39	0,15	-	-	-	-	-	-	3142,32	-
CHL1s	-	390,48	-	913,05	-	226,19	217,95	50,57	48,12

Runtime Results for K=6 in Seconds

File	**Gap [%]**	**Simple**				**CSA**			
		naive		strip		strip		all	
		K=9	K=6	K=9	K=6	K=9	K=6	K=9	K=6
CHL2s	-	0,08	0,07	0,04	0,05	0,66	0,67	0,31	0,31
CHL3s	46,41	-	-	-	-	-	-	-	-
CHL4s	36,83	-	-	-	-	-	-	-	-
CHL5	-	0,13	0,13	0,13	0,13	0,94	0,65	0,21	0,14
CHL6	-	-	-	156,01	172,66	1760,03	1956,82	210,04	258,74
CHL7	-	-	-	-	-	-	-	509,39	500,50
CU1	-	6,07	6,60	3,50	4,76	0,27	0,27	0,18	0,18
CU2	-	12,17	11,94	3,20	3,84	0,87	0,87	0,57	0,57
CU3	-	246,79	275,17	15,73	17,69	1,99	1,97	1,25	1,35
CU4	-	27,15	26,57	3,86	3,88	4,14	4,15	2,69	2,67
CU5	-	32,99	34,57	0,61	0,63	1,71	1,67	1,25	1,26
CU6	-	4,87	5,36	0,88	0,90	0,15	0,16	0,13	0,14
CU7	-	1,83	1,78	0,70	0,76	0,87	0,86	0,52	0,52
CU8	-	1,39	1,49	0,51	0,52	0,50	0,39	0,36	0,28
CU9	-	7,90	8,06	0,71	0,75	0,09	0,08	0,08	0,07
CU10	-	1185,11	2826,87	877,70	966,27	60,42	61,54	25,44	25,07
CU11	-	-	-	-	-	2325,43	2130,03	701,70	671,50
Hchl3s	0,01	0,67	0,66	6,48	6,72	-	-	1082,30	1180,34
Hchl4s	1,36	149,21	156,56	138,58	193,66	-	-	-	-
Hchl5s	0,30	-	-	-	-	-	-	-	-
Hchl6s	-	-	-	624,83	-	929,97	964,83	205,38	196,00
Hchl7s	-	-	-	-	-	2220,31	2763,82	617,08	788,25
Hchl8s	1,87	189,18	161,08	218,32	166,96	-	-	-	-
Hs	-	0,20	0,20	0,04	0,05	461,62	188,15	0,35	0,02
HZ1s	-	0,01	0,01	0,01	0,01	-	-	1,14	1,13
M1	-	0,31	0,35	0,07	0,08	0,78	0,78	0,07	0,07
M2	-	0,81	0,79	0,12	0,12	1,39	1,38	0,36	0,36
M3	-	0,12	0,13	0,03	0,03	0,11	0,11	0,06	0,07
M4s	-	0,17	0,17	0,06	0,07	0,98	0,99	0,10	0,10
M5	-	0,42	0,42	0,09	0,10	0,96	0,95	0,09	0,09
OF1	-	0,06	0,06	0,04	0,04	0,10	0,09	0,06	0,07
OF2	-	0,40	0,39	0,08	0,08	0,55	0,33	0,30	0,18
STS2s	-	0,69	0,75	0,46	0,53	1,35	1,19	0,88	0,82
STS4s	-	1507,97	-	-	-	157,32	31,83	25,60	14,83
UU1s	-	2,51	2,74	0,47	0,50	0,36	0,37	0,20	0,20
UU2s	-	5,35	5,97	5,75	6,82	0,06	0,06	0,05	0,05
UU3s	-	0,50	0,52	0,92	1,05	0,27	0,25	0,20	0,17
UU4s	-	4,44	4,82	18,01	19,45	2,77	2,78	1,58	1,60
UU5s	-	24,60	26,19	5,19	5,63	0,51	0,52	0,37	0,37

Runtime Results for K=6 in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	K=6	K=9	K=6	K=9	K=6	K=9	K=6
UU6s	-	13,36	17,27	0,91	1,15	0,11	0,09	0,10	0,08
UU7s	-	394,00	-	27,76	29,36	11,16	11,03	5,60	5,64
UU8s	-	126,04	143,63	72,42	77,86	1,41	1,43	0,85	0,86
UU9	-	26,67	28,49	7,78	8,43	0,39	0,38	0,31	0,30
UU10	-	173,49	185,37	18,57	20,62	12,78	12,94	5,44	5,42
W	-	0,34	0,37	0,11	0,12	0,70	0,61	0,33	0,20

Table 4.6 shows the runtime of the algorithm with the number of stages K limited to six, the gap to the optimal score and the runtime compared to $K = 9$. Only eight instances could not be solved within the six stage requirement, five of which couldn't be solved optimally with $K = 9$. The score deviation for stages whose optimum requires more than six stages is less than one percent for all except *Hchl8s*, which could not be solved optimally anyway.

There is a disparity between the Simple algorithm and the CSA algorithm. The runtimes of the Simple algorithm increased, while the runtimes of the CSA algorithm generally decreased or remained roughly the same. The instances with an optimal solution above six stages generally show increased runtimes for the CSA algorithm. It might be useful to attempt to solve an instance with a higher K than is desired, if the amount of stages required is unknown.

4.3.2 4 Stages

Restricting K to four means that some instances will achieve a lower score than with a higher K. We observe the changes in runtime and how much the score deviates.

Table 4.7: Runtime Results for K=4 in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	K=4	K=9	K=4	K=9	K=4	K=9	K=4
2s	1,37	7,38	8,68	2,50	5,57	1,10	6,60	0,44	1,38
3s	-	0,42	0,44	0,22	0,22	0,65	0,09	0,32	0,06
A1s	-	0,46	0,41	0,15	0,14	0,03	0,02	0,03	0,02
A2s	2,15	1,14	1,16	0,75	0,73	0,18	0,43	0,13	0,28
A3	0,75	3,46	3,35	3,28	3,10	42,36	48,11	14,70	16,47
A4	1,57	146,80	154,94	81,18	79,58	156,02	353,20	52,87	99,98
A5	0,07	136,97	141,16	45,27	77,02	297,03	120,70	79,25	133,67
ATP37	0,96	-	-	-	-	-	-	3096,56	-
ATP39	0,32	-	-	-	-	-	-	3142,32	-
CHL1s	0,45	390,48	403,12	913,05	1853,62	226,19	316,55	50,57	132,51

Runtime Results for $K=4$ in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		$K=9$	$K=4$	$K=9$	$K=4$	$K=9$	$K=4$	$K=9$	$K=4$
CHL2s	-	0,08	0,07	0,04	0,05	0,66	0,58	0,31	0,27
CHL3s	46,41	-	-	-	-	-	-	-	-
CHL4s	32,72	-	-	-	-	-	-	-	-
CHL5	-	0,13	0,12	0,13	0,11	0,94	0,28	0,21	0,11
CHL6	-	-	-	156,01	345,79	1760,03	2580,48	210,04	405,05
CHL7	0,25	-	-	-	-	-	-	509,39	1664,15
CU1	-	6,07	7,04	3,50	3,69	0,27	0,25	0,18	0,17
CU2	-	12,17	12,38	3,20	3,32	0,87	0,84	0,57	0,56
CU3	-	246,79	292,77	15,73	17,86	1,99	2,13	1,25	1,46
CU4	0,16	27,15	29,05	3,86	3,97	4,14	5,10	2,69	3,25
CU5	-	32,99	35,43	0,61	0,62	1,71	1,63	1,25	1,21
CU6	-	4,87	4,98	0,88	0,90	0,15	0,15	0,13	0,14
CU7	-	1,83	2,04	0,70	0,74	0,87	0,88	0,52	0,52
CU8	0,14	1,39	1,39	0,51	0,46	0,50	0,40	0,36	0,29
CU9	-	7,90	7,99	0,71	0,72	0,09	0,09	0,08	0,07
CU10	0,04	1185,11	1210,11	877,70	896,33	60,42	57,29	25,44	25,67
CU11	-	-	-	-	-	2325,43	-	701,70	1127,25
Hchl3s	0,05	0,67	0,64	6,48	6,10	-	-	1082,30	416,28
Hchl4s	1,36	149,21	133,78	138,58	114,83	-	-	-	-
Hchl5s	1,37	-	-	-	-	-	-	-	-
Hchl6s	-	-	-	624,83	756,19	929,97	1521,51	205,38	326,20
Hchl7s	0,14	-	-	-	-	2220,31	-	617,08	1532,46
Hchl8s	3,18	189,18	87,11	218,32	90,90	-	-	-	2507,97
Hs	-	0,20	0,18	0,04	0,04	461,62	3,52	0,35	0,02
HZ1s	-	0,01	0,01	0,01	0,01	-	-	1,14	1,07
M1	-	0,31	0,35	0,07	0,07	0,78	0,37	0,07	0,07
M2	0,84	0,81	0,70	0,12	0,12	1,39	0,94	0,36	0,33
M3	-	0,12	0,12	0,03	0,03	0,11	0,11	0,06	0,06
M4s	-	0,17	0,15	0,06	0,07	0,98	0,49	0,10	0,09
M5	-	0,42	0,41	0,09	0,10	0,96	0,47	0,09	0,08
OF1	0,88	0,06	0,06	0,04	0,05	0,10	0,10	0,06	0,06
OF2	-	0,40	0,36	0,08	0,07	0,55	0,54	0,30	0,27
STS2s	0,19	0,69	0,74	0,46	0,51	1,35	1,24	0,88	0,88
STS4s	0,19	1507,97	1560,83	-	-	157,32	23,78	25,60	22,38
UU1s	-	2,51	2,76	0,47	0,46	0,36	0,37	0,20	0,21
UU2s	-	5,35	5,92	5,75	5,89	0,06	0,06	0,05	0,05
UU3s	0,62	0,50	0,52	0,92	0,94	0,27	0,48	0,20	0,30
UU4s	0,14	4,44	4,75	18,01	18,40	2,77	2,68	1,58	1,53
UU5s	0,00	24,60	23,84	5,19	5,25	0,51	0,48	0,37	0,35

Runtime Results for K=4 in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	*K=4*	*K=9*	*K=4*	*K=9*	*K=4*	*K=9*	*K=4*
UU6s	-	13,36	14,20	0,91	1,03	0,11	0,09	0,10	0,08
UU7s	-	394,00	384,26	27,76	29,80	11,16	9,02	5,60	4,81
UU8s	-	126,04	126,50	72,42	74,02	1,41	1,32	0,85	0,83
UU9	-	26,67	26,11	7,78	7,96	0,39	0,36	0,31	0,29
UU10	-	173,49	194,37	18,57	18,95	12,78	10,91	5,44	5,55
W	0,77	0,34	0,35	0,11	0,10	0,70	0,56	0,33	0,21

Table 4.7 shows the runtime of the algorithm with the number of stages K limited to four, the gap to the optimal score and the runtime compared to $K = 9$.

Instance *CHL4s* still can not be solved within the timeout, but is the only instance that shows a, relatively sizeable, improvement with a lower K. With this the best found score of the CSA algorithm for *CHL4s* rises to 9373 (Optimum: 13923).

The biggest relative change is *CHL4s* with a 6,4% gain. The biggest relative change on an instance that can be solved within the timeout is *A2s* with a 2,15% loss.

The runtimes show a slight increase for most, but not all, instances that can be solved optimally within four stages. Overall restricting the stages does not show any distinct advantage.

4.3.3 3 Stages

Three-staged cutting patterns represent a widely used category in industry. In general the three-staged patterns make a good balance between material utilization and cutting complexity [6]. 13 out of the 55 instances can be solved optimally with $K = 3$. As before we observer runtime changes and score deviation.

Table 4.8: Runtime Results for K=3 in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*
2s	5,40	7,38	159,53	2,50	114,81	1,10	277,43	0,44	377,94
3s	0,77	0,42	9,25	0,22	2,82	0,65	4,10	0,32	2,00
A1s	-	0,46	5,67	0,15	0,56	0,03	0,20	0,03	0,19
A2s	2,55	1,14	10,61	0,75	2,54	0,18	2,47	0,13	1,82
A3	0,75	3,46	10,51	3,28	20,18	42,36	72,18	14,70	38,91
A4	2,59	146,80	224,35	81,18	110,42	156,02	-	52,87	708,14
A5	1,59	136,97	878,06	45,27	332,08	297,03	-	79,25	1351,08
ATP37	1,29	-	-	-	-	-	-	3096,56	-
ATP39	1,09	-	-	-	-	-	-	3142,32	-
CHL1s	0,48	390,48	-	913,05	-	226,19	1746,67	50,57	598,35

Runtime Results for *K=3* in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*
CHL2s	-	0,08	0,34	0,04	0,30	0,66	9,12	0,31	4,08
CHL3s	46,41	-	-	-	-	-	-	-	-
CHL4s	36,83	-	-	-	-	-	-	-	-
CHL5	1,03	0,13	0,74	0,13	0,75	0,94	3,36	0,21	1,14
CHL6	3,16	-	-	156,01	-	1760,03	-	210,04	-
CHL7	3,02	-	-	-	-	-	-	509,39	-
CU1	0,15	6,07	30,78	3,50	10,05	0,27	2,27	0,18	1,74
CU2	-	12,17	111,39	3,20	5,59	0,87	5,85	0,57	4,77
CU3	0,54	246,79	256,19	15,73	2314,40	1,99	20,09	1,25	14,73
CU4	0,23	27,15	3414,75	3,86	35,98	4,14	37,40	2,69	24,62
CU5	1,05	32,99	1484,75	0,61	67,27	1,71	49,41	1,25	27,29
CU6	-	4,87	617,04	0,88	168,66	0,15	1,15	0,13	1,04
CU7	0,41	1,83	6,92	0,70	2,75	0,87	4,66	0,52	3,15
CU8	0,14	1,39	23,26	0,51		0,50	4,16	0,36	3,31
CU9	2,24	7,90	68,48	0,71	6,20	0,09	7,29	0,08	4,00
CU10	0,04	1185,11	-	877,70	1375,62	60,42	165,01	25,44	90,73
CU11	0,80	-	-	-	901,95	2325,43	-	701,70	-
Hchl3s	2,89	0,67	-	6,48	-	-	-	1082,30	-
Hchl4s	5,63	149,21	-	138,58	-	-	-	-	-
Hchl5s	3,17	-	-	-	-	-	-	-	-
Hchl6s	0,35	-	-	624,83	834,14	929,97	-	205,38	3483,90
Hchl7s	2,05	-	-	-	-	2220,31	-	617,08	-
Hchl8s	8,89	189,18	426,98	218,32	445,88	-	-	-	-
Hs	1,75	0,20	4,83	0,04	1,29	461,62	8,05	0,35	1,57
HZ1s	-	0,01	0,07	0,01	0,06	-	165,45	1,14	8,66
M1	-	0,31	2,32	0,07	1,31	0,78	1,86	0,07	0,52
M2	0,84	0,81	1,95	0,12	0,38	1,39	2,32	0,36	1,35
M3	1,47	0,12	4,85	0,03	1,33	0,11	2,01	0,06	1,70
M4s	-	0,17	1,84	0,06	1,15	0,98	2,46	0,10	0,74
M5	-	0,42	2,00	0,09	1,13	0,96	2,33	0,09	0,71
OF1	0,88	0,06	0,86	0,04	0,93	0,10	0,49	0,06	0,37
OF2	6,51	0,40	2,85	0,08	1,09	0,55	11,64	0,30	4,23
STS2s	0,92	0,69	170,73	0,46	-	1,35	458,87	0,88	263,58
STS4s	3,48	1507,97	-	-	383,83	157,32	-	25,60	-
UU1s	0,68	2,51	34,18	0,47	4,73	0,36	2,87	0,20	1,75
UU2s	-	5,35	39,05	5,75	22,95	0,06	0,61	0,05	0,52
UU3s	1,22	0,50	11,75	0,92	7,23	0,27	4,30	0,20	2,86
UU4s	0,14	4,44	899,23	18,01	120,96	2,77	18,92	1,58	12,49
UU5s	0,00	24,60	402,38	5,19	158,56	0,51	3,54	0,37	2,87

Runtime Results for *K=3* in Seconds

File	**Gap** [%]	**Simple**				**CSA**			
		naive		strip		strip		all	
		K=9	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*	*K=9*	*K=3*
UU6s	-	13,36	213,41	0,91	15,06	0,11	0,72	0,10	0,66
UU7s	0,23	394,00	2631,99	27,76	1076,39	11,16	151,76	5,60	65,52
UU8s	-	126,04	1312,69	72,42	158,58	1,41	7,01	0,85	5,35
UU9	-	26,67	1923,52	7,78	309,98	0,39	2,78	0,31	2,39
UU10	0,21	173,49	-	18,57	749,79	12,78	101,54	5,44	52,02
W	3,60	0,34	2,16	0,11	3,84	0,70	5,29	0,33	2,33

Table 4.8 shows the runtime of the algorithm with the number of stages K limited to three, the gap to the optimal score and the runtime compared to $K = 9$.

Again the runtimes increase overall. Several instances that were solvable with a higher stage limit could not be solved within the timeout. All the optimally solvable instances show a notably increased runtime. Instance *HZ1s*, which has a very simple two staged optimal pattern, requires nearly eight seconds as opposed to one second before.

Another striking result is that instance *CU11*, which is optimally solvable with three stages, does not find an optimal solution within the timeout. Using $K = 9$ the optimal solution was found within 12 minutes.

From all instances that finished within the timeout *OF2* and *2s* show the worst decline in score.

4.3.4 2 Stages

As with a three stage requirement, $K = 2$ represents a widely used restriction in industry. Only eight instances have an optimal pattern with a two stage requirement.

Table 4.9: Runtime Results for K=2 in Seconds

File	**Gap** [%]	**Simple**				**CSA**			
		naive		strip		strip		all	
		K=9	*K=2*	*K=9*	*K=2*	*K=9*	*K=2*	*K=9*	*K=2*
2s	17,42	7,38	4,01	2,50	4,09	1,10	-	0,44	206,88
3s	11,61	0,42	3,52	0,22	2,48	0,65	16,05	0,32	2,79
A1s	-	0,46	1,69	0,15	1,77	0,03	0,14	0,03	0,13
A2s	4,61	1,14	1,10	0,75	1,21	0,18	0,49	0,13	0,39
A3	1,30	3,46	6,32	3,28	6,28	42,36	6,08	14,70	4,08
A4	7,46	146,80	3,31	81,18	3,41	156,02	135,14	52,87	24,73
A5	5,46	136,97	7,76	45,27	8,40	297,03	962,96	79,25	306,60
ATP37	1,77	-	-	-	-	-	-	3096,56	-
ATP39	1,44	-	-	-	-	-	-	3142,32	-
CHL1s	2,90	390,48	37,27	913,05	37,15	226,19	2798,38	50,57	1160,77
CHL2s	3,57	0,08	0,41	0,04	0,41	0,66	1,52	0,31	0,64

Runtime Results For K=2 in Seconds

File	Gap [%]	Simple				CSA			
		naive		strip		strip		all	
		K=9	K=2	K=9	K=2	K=9	K=2	K=9	K=2
CHL3s	41,35	-	-	-	-	-	-	-	-
CHL4s	25,23	-	-	-	3345,35	-	-	-	-
CHL5	6,92	0,13	0,24	0,13	0,22	0,94	1,70	0,21	0,61
CHL6	2,58	-	288,73	156,01	277,36	1760,03	-	210,04	3455,41
CHL7	3,23	-	1551,92	-	1708,06	-	-	509,39	-
CU1	0,61	6,07	2,00	3,50	2,01	0,27	0,58	0,18	0,55
CU2	-	12,17	17,49	3,20	18,49	0,87	16,01	0,57	5,11
CU3	0,69	246,79	109,42	15,73	106,45	1,99	15,88	1,25	14,50
CU4	1,03	27,15	80,82	3,86	80,00	4,14	49,12	2,69	14,97
CU5	3,56	32,99	89,86	0,61	94,99	1,71	140,82	1,25	19,99
CU6	-	4,87	174,85	0,88	165,37	0,15	6,26	0,13	4,55
CU7	1,80	1,83	2,92	0,70	3,25	0,87	1,25	0,52	0,90
CU8	4,13	1,39	14,61	0,51	16,36	0,50	9,89	0,36	3,49
CU9	4,44	7,90	3,92	0,71	3,93	0,09	5,58	0,08	2,24
CU10	1,36	1185,11	593,76	877,70	596,90	60,42	192,47	25,44	80,54
CU11	1,74	-	3068,38	-	2820,26	2325,43	-	701,70	2422,49
Hchl3s	3,98	0,67	2,54	6,48	2,67	-	-	1082,30	2483,26
Hchl4s	5,78	149,21	2,24	138,58	2,21	-	-	-	1423,19
Hchl5s	3,51	-	-	-	-	-	-	-	-
Hchl6s	4,28	-	72,26	624,83	80,91	929,97	-	205,38	933,75
Hchl7s	1,52	-	1657,77	-	1733,48	2220,31	-	617,08	-
Hchl8s	21,62	189,18	0,52	218,32	0,52	-	2597,75	-	68,07
Hs	1,75	0,20	0,09	0,04	0,11	461,62	0,58	0,35	0,55
HZ1s	-	0,01	0,28	0,01	0,28	-	9,55	1,14	4,39
M1	-	0,31	0,31	0,07	0,30	0,78	0,21	0,07	0,18
M2	1,37	0,81	0,20	0,12	0,21	1,39	0,38	0,36	0,36
M3	3,06	0,12	0,40	0,03	0,40	0,11	1,20	0,06	0,58
M4s	-	0,17	0,29	0,06	0,30	0,98	0,23	0,10	0,24
M5	-	0,42	0,31	0,09	0,31	0,96	0,20	0,09	0,19
OF1	0,88	0,06	0,17	0,04	0,18	0,10	0,15	0,06	0,12
OF2	10,89	0,40	0,23	0,08	0,25	0,55	2,67	0,30	0,98
STS2s	3,55	0,69	142,77	0,46	140,77	1,35	526,08	0,88	141,37
STS4s	3,00	1507,97	21,66	-	23,13	157,32	690,06	25,60	240,08
UU1s	2,97	2,51	3,34	0,47	3,18	0,36	2,05	0,20	1,04
UU2s	1,94	5,35	8,20	5,75	8,21	0,06	1,92	0,05	1,39
UU3s	5,19	0,50	4,04	0,92	4,10	0,27	4,80	0,20	2,03
UU4s	1,56	4,44	38,53	18,01	40,55	2,77	9,38	1,58	5,72
UU5s	1,92	24,60	94,52	5,19	97,52	0,51	17,88	0,37	10,16
UU6s	0,22	13,36	16,32	0,91	16,51	0,11	0,56	0,10	0,54

Runtime Results For *K=2* in Seconds

File	**Gap [%]**	**Simple**				**CSA**			
		naive		strip		strip		all	
		K=9	*K=2*	*K=9*	*K=2*	*K=9*	*K=2*	*K=9*	*K=2*
UU7s	1,19	394,00	159,39	27,76	154,32	11,16	15,84	5,60	9,75
UU8s	0,57	126,04	111,34	72,42	109,86	1,41	2,04	0,85	1,91
UU9	2,20	26,67	162,47	7,78	156,18	0,39	15,83	0,31	7,62
UU10	2,66	173,49	777,99	18,57	828,19	12,78	169,52	5,44	45,07
W	9,22	0,34	0,26	0,11	0,27	0,70	0,74	0,33	0,61

Table 4.9 shows the runtime of the algorithm with the number of stages K limited to two, the gap to the optimal score and the runtime compared to $K = 9$.

Several instances that weren't solvable with $K = 3$ finished within the timeout again. Overall the runtimes still show a notable increase, however the times decreased when compared to $K = 3$.

CHL3s and *CHL4s* show an increase in score again. This is due to the algorithm creating bigger compounds faster, since it is limited to two stages.

Both score and runtime perform worse than $K = 9$. The runtime difference between $K = 3$ and $K = 2$ suggests that the lower stage limit can be used in cases where three-staged patterns take too long to compute. This is as a tradeoff to solution quality, since the three-staged patterns show a much better score, if applicable.

4.4 Comparison to a Beam-Search Approach Utilizing an Insertion Heuristic

The K-2CSV Framework of the *Algorithms and Complexity Group* contains a Beam-Search algorithm, described in [15], that utilizes an adapted insertion heuristic from Fleszar [18]. The algorithm creates a full solution for the problem, containing multiple sheet. No sheet is fixed until the algorithm finishes. We therefore take the first sheet of the complete solution for the comparison, since it is bound to have the highest value due to the normal form. All instances only contain exactly one stock sheet type, meaning that we do not have to consider multiple sheet types for the comparison.

Table 4.10: Beam-Search Comparison

File	**Gap [%]**		**Score**		**Runtime [s]**	
	CSA	*Optimum*	*CSA*	*Beam-Search*	*CSA*	*Beam-Search*
2s	1,37	1,37	2778	2740	0,44	0,01
3s	20,32	20,32	2721	2168	0,32	0,07
A1s	6,51	6,51	2950	2758	0,03	0,08
A2s	25,86	25,86	3535	2621	0,13	0,06
A3	13,26	13,26	5451	4728	14,70	0,05
A4	8,14	8,14	6179	5676	52,87	0,05

Beam-Search Comparison

File	**Gap [%]**		**Score**		**Runtime [s]**	
	CSA	*Optimum*	*CSA*	*Beam-Search*	*CSA*	*Beam-Search*
A5	15,73	15,73	12985	10943	79,25	0,01
ATP37	15,54	15,54	387276	327094	3096,56	0,20
ATP39	6,93	6,93	268750	250122	3142,32	0,14
CHL1s	11,42	11,42	13099	11603	50,57	0,08
CHL2s	9,45	9,45	3279	2969	0,31	0,05
CHL3s	−86,59	-	3967	7402	-	0,05
CHL4s	−58,30	-	8801	13932	-	0,05
CHL5	12,05	12,05	390	343	0,21	0,05
CHL6	7,90	7,90	16869	15536	210,04	0,08
CHL7	37,27	37,27	16881	10590	509,39	0,08
CU10	9,03	9,03	773772	703924	25,44	0,01
CU11	8,09	8,09	924696	849844	701,70	0,02
CU1	7,37	7,37	12330	11421	0,18	0,01
CU2	14,73	14,73	26100	22255	0,57	0,01
CU3	8,14	8,14	16723	15362	1,25	0,17
CU4	3,99	3,99	99495	95527	2,69	0,14
CU5	4,42	4,42	173364	165708	1,25	0,19
CU6	8,58	8,58	158572	144965	0,13	0,16
CU7	7,69	7,69	247150	228145	0,52	0,08
CU8	11,91	11,91	433331	381716	0,36	0,11
CU9	15,83	15,83	657055	553020	0,08	0,09
Hchl3s	7,92	7,92	12215	11248	1082,30	0,06
Hchl4s	24,02	24,87	11859	9011	-	0,05
Hchl5s	6,70	6,99	45223	42192	-	0,07
Hchl6s	23,17	23,17	61040	46897	205,38	0,07
Hchl7s	8,13	8,13	63112	57981	617,08	0,10
Hchl8s	14,46	14,93	906	775	-	0,05
Hs	2,92	2,92	12348	11988	0,35	0,05
HZ1s	-	-	5226	5226	1,14	0,01
M1	4,60	4,60	15024	14333	0,07	0,01
M2	5,53	5,53	73176	69132	0,36	0,01
M3	3,83	3,83	142817	137352	0,06	0,01
M4s	3,99	3,99	265768	255172	0,10	0,07
M5	5,15	5,15	577882	548098	0,09	0,07
OF1	7,31	7,31	2737	2537	0,06	0,05
OF2	13,83	13,83	2690	2318	0,30	0,05
STS2s	8,17	8,17	4653	4273	0,88	0,09
STS4s	12,58	12,58	9770	8541	25,60	0,01
UU10	6,19	6,19	11955852	11215805	5,44	0,06
UU1s	9,22	9,22	242919	220520	0,20	0,01

Beam-Search Comparison

File	Gap [%]		Score		Runtime [s]	
	CSA	*Optimum*	*CSA*	*Beam-Search*	*CSA*	*Beam-Search*
UU2s	7,37	7,37	595288	551418	0,05	0,01
UU3s	15,72	15,72	1072764	904106	0,20	0,01
UU4s	13,78	13,78	1179050	1016542	1,58	0,29
UU5s	11,97	11,97	1868999	1645257	0,37	0,46
UU6s	18,97	18,97	2950760	2390917	0,10	0,03
UU7s	17,83	17,83	2930654	2408113	5,60	0,48
UU8s	8,26	8,26	3959352	3632198	0,85	0,55
UU9	14,95	14,95	6100692	5188841	0,31	0,54
W	17,90	17,90	2721	2234	0,33	0,06

Table 4.10 compares the results of the Beam-Search to the CSA algorithm with all improvements. Both algorithms were executed with a stage limit of $K = 9$. The score derivation between the algorithms is shown in the *absolute* column. The *CSA-Gap* column shows how close the Beam-Search came to the score of the CSA algorithm, while the *Optimum-Gap* column shows how close the Beam-Search came to the optimal value.

The runtime column shows that the heuristic Beam-Search approach is very fast compared to the exact Branch-and-Bound algorithm. Even though the Beam-Search has to calculate all sheets, this is to be expected since its search space is much smaller.

Average solution value of the Beam-Search is 88.9% of the optimal value. The worst instance for it is *A2s* where only 74,14% of the optimal value where achieved, which is rather surprising considering *A2s*' optimal pattern is not very complex. The amount of unused element types in the optimal solution makes it harder for the Beam-Search to find a good pattern for only one sheet. Opposed to that are *CHL3s* and *CHL4s* which the CSA algorithm could not solve within the timeout, but the Beam-Search solved optimally. Both of these instances are a worst-caste situation for the Branch-and-Bound algorithm: Many small elements on a very large sheet. The optimal solution for those instances only cover 39% and 29% of the sheets area.

Overall the Beam-Search finds acceptable solutions very quickly, however there is room for improvement since the average solution quality only reaches about 90% of the best one. The CAS algorithm is superior when the stock sheet material is expensive, since it utilizes the available area better. For only partially filled sheets, or when a quick result is needed and the material used is cheap, the Beam-Search or another heuristic should be preferred.

4.5 Comparison of the Results

In this section we give an overview over the gap between the optimal score and the CSA algorithm with different K restrictions as well as the Beam-Search.

Table 4.11: Comparison of Score Gaps to Optimum

File	**Gap to Optimum [%]**					
	$K = 9$	$K = 6$	$K = 4$	$K = 3$	$K = 2$	**Beam**
2s	-	-	1,37	5,40	17,42	1,37
3s	-	-	-	0,77	11,61	20,32
A1s	-	-	-	-	-	6,51
A2s	-	-	2,15	2,55	4,61	25,86
A3	-	-	0,75	0,75	1,30	13,26
A4	-	0,99	1,57	2,59	7,46	8,14
A5	-	-	0,07	1,59	5,46	15,73
ATP37	-	-	0,96	1,29	1,77	15,54
ATP39	-	0,15	0,32	1,09	1,44	6,93
CHL1s	-	-	0,45	0,48	2,90	11,42
CHL2s	-	-	-	-	3,57	9,45
CHL3s	46,41	46,41	46,41	46,41	41,35	-
CHL4s	36,83	36,83	32,72	36,83	25,23	-
CHL5	-	-	-	1,03	6,92	12,05
CHL6	-	-	-	3,16	2,58	7,90
CHL7	-	-	0,25	3,02	3,23	37,27
CU1	-	-	-	0,15	0,61	7,37
CU2	-	-	-	-	-	14,73
CU3	-	-	-	0,54	0,69	8,14
CU4	-	-	0,16	0,23	1,03	3,99
CU5	-	-	-	1,05	3,56	4,42
CU6	-	-	-	-	-	8,58
CU7	-	-	-	0,41	1,80	7,69
CU8	-	-	0,14	0,14	4,13	11,91
CU9	-	-	-	2,24	4,44	15,83
CU10	-	-	0,04	0,04	1,36	9,03
CU11	-	-	-	0,80	1,74	8,09
Hchl3s	-	0,01	0,05	2,89	3,98	7,92
Hchl4s	1,13	1,36	1,36	5,63	5,78	24,87
Hchl5s	0,30	0,30	1,37	3,17	3,51	6,99
Hchl6s	-	-	-	0,35	4,28	23,17
Hchl7s	-	-	0,14	2,05	1,52	8,13
Hchl8s	0,55	1,87	3,18	8,89	21,62	14,93
Hs	-	-	-	1,75	1,75	2,92
HZ1s	-	-	-	-	-	-
M1	-	-	-	-	-	4,60
M2	-	-	0,84	0,84	1,37	5,53
M3	-	-	-	1,47	3,06	3,83
M4s	-	-	-	-	-	3,99
M5	-	-	-	-	-	5,15

Comparison of Score Gaps to Optimum

File	**Gap to Optimum** [%]					
	$K = 9$	$K = 6$	$K = 4$	$K = 3$	$K = 2$	**Beam**
OF1	-	-	0,88	0,88	0,88	7,31
OF2	-	-	-	6,51	10,89	13,83
STS2s	-	-	0,19	0,92	3,55	8,17
STS4s	-	-	0,19	3,48	3,00	12,58
UU1s	-	-	-	0,68	2,97	9,22
UU2s	-	-	-	-	1,94	7,37
UU3s	-	-	0,62	1,22	5,19	15,72
UU4s	-	-	0,14	0,14	1,56	13,78
UU5s	-	-	0,00	0,00	1,92	11,97
UU6s	-	-	-	-	0,22	18,97
UU7s	-	-	-	0,23	1,19	17,83
UU8s	-	-	-	-	0,57	8,26
UU9	-	-	-	-	2,20	14,95
UU10	-	-	-	0,21	2,66	6,19
W	-	-	0,77	3,60	9,22	17,90

Table 4.11 shows the gaps to the optimal value for different stage restrictions with the CSA algorithm and the beam search.

The table shows that less allowed stages result in less optimal patterns, which is to be expected considering it decreases pattern complexity and therefore limits the possibility to fill gaps. Considering only instances that could be solved within the timeout, the gap mostly stays below five percent for low values of K. It also becomes apparent that the CSA algorithm outperforms the Beam-Search on nearly every instance with respect to the score.

CHAPTER 5

Summary, Conclusion and Future Work

In the course of this work we presented two bottom-up Branch-and-Bound approaches to solve the K-2CSV problem. The basic principle behind the algorithms is to combine patterns together to find an optimal cutting pattern with the least possible space wasted for a given stock sheet. All elements used have a demand, which represents a constraint on how often it can occur, and each element type's profit is equal to the area it requires. All patterns are in normal form, aligned in the top left corner, sorted by size. All cuts made in the pattern are guillotine cuts. This ensures that the patterns can be processed in industrial machinery and usually is a standard requirement. The algorithms were integrated in a framework to solve instances across multiple sheets, solving each sheet separately. This does not guarantee an optimal solution over all sheets but for each individual sheet.

Since this type of problem can take very long the option to abort after a timeout is presented. In this case the best known solution at the moment of the timeout is taken. Should a timeout occur all guarantees for optimality are forfeited.

The first algorithm is a basic implementation called Simple algorithm. The basic premise is to take a pattern, starting with a single element, and combine it with each element in every way possible. Then continue on to do this recursively. The big difference to other Branch-and-Bound approaches is that this algorithm only adds a single element to the existing pattern, while other algorithms typically combine with more complex patterns. This also requires it to take the order in which the elements are combined into account.

The second Cutting Stock Algorithm (CSA) is based on the *Modified Viswanathan and Bagchi* (MVB) algorithm. We keep a list of all valid patterns encountered so far and create new ones by taking the current given pattern and combining it vertically and horizontally with each of those and itself.

We implemented several improvements to increase runtime performance. The strip upper-bound provided the best improvement. It takes the remaining space on the sheet, divides it into strips and then calculates which combination of elements would provide the best possible gain. The results show that this upper bound outperforms other bounds, unless the total runtime is short enough to be dominated by the setup-time to calculate the combinations required for the strips (< 1 second).

The CSA algorithm has been improved with best-first-traversal (BFS), detection of dominated patterns and advanced detection of duplicate patterns. BFS and dominated patterns do not provide a big improvement on their own however they still make sense in the overall picture and provide means of reducing the search space not provided by other measures. This mostly helps reducing the runtime in combination with other improvements.

The results show that the advanced duplicate detection brings notable improvements. However it is apparent that the inherent property to not add constraint-violating patterns to the branching tree of the CSA algorithm plays an important role. While the improvement from saved memory operations is obvious, this property also drastically reduces the combinations to check for the advanced duplicate detection. Without it the additional computations required can outweigh the gains from the advanced detection.

Overall the computational results show that the CSA algorithm is a good general approach to solve the K-2CSV problem.

A thing to note is that the stage limit K can impact the runtime negatively even when the optimal solution is within the limit. Slow instances might be solvable faster by increasing K, if their optimal solution turns out to lie within the original K.

Furthermore it is noticeable that the general impact of K on the achievable score is relatively low ($\leq 2\%$). In rare cases it is possible to find a better solution with a lower K in combination with a reasonable timeout.

5.1 Future Work

The Simple algorithm will not be able to compete with the CSA algorithm on most patterns, however additional work may be beneficial to obtain an algorithm that can complement the CSA algorithm on unfavourable instances that profit of the depth-first nature of the Simple algorithm.

Several properties of the CSA algorithm may be adapted to fit the Simple algorithm.

Premature checks of patterns before adding them to the branching tree, advanced duplicate detection and possibly different data structures are only a few.

The current strip-bound is based on an unconstrained problem. A better upper bound that also takes element demands into account is a promising area of research. As the results show, the upper bound has the biggest impact on performance.

One could also try to find a better and/or faster means to reduce the search space like:

- a more sophisticated duplicate detection for patterns that covers more cases.
- a different vector of attack to find dominated or obsolete patterns.
- earlier detection of promising patterns to process them first.
- detecting when the sheet cannot be fully filled because the elements are too small and terminating the algorithm with the first acceptable solution that uses all elements.

Bibliography

[1] R. Alvarez-Valdés, A. Parajón, et al. A tabu search algorithm for large-scale guillotine (un) constrained two-dimensional cutting problems. *Computers & Operations Research*, 29(7):925–947, 2002.

[2] J. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[3] C. Charalambous and K. Fleszar. A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts. *Computers & Operations Research*, 38(10):1443–1451, 2011.

[4] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25(1):30–44, 1977.

[5] G. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. 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.

[6] Y. Cui. A new dynamic programming procedure for three-staged cutting patterns. *Journal of global optimization*, 55(2):349–357, 2013.

[7] Y. Cui and B. Huang. Heuristic for constrained t-shape cutting patterns of rectangular pieces. *Computers & Operations Research*, 39(12):3031–3039, 2012.

[8] 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.

[9] Y. Cui and X. Zhang. Two-stage general block patterns for the two-dimensional cutting problem. *Computers & Operations Research*, 34(10):2882–2893, 2007.

[10] Y.-P. Cui, Y. Cui, and T. Tang. Sequential heuristic for the two-dimensional bin-packing problem. *European Journal of Operational Research*, 240(1):43–53, 2015.

[11] Y.-P. Cui, Y. Cui, T. Tang, and W. Hu. Heuristic for constrained two-dimensional three-staged patterns. *Journal of the Operational Research Society*, 66(4):647–656, 2014.

[12] V.-D. Cung, M. 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.

[13] M. Dolatabadi, A. Lodi, and M. Monaci. Exact algorithms for the two-dimensional guillotine knapsack. *Computers & Operations Research*, 39(1):48–53, 2012.

[14] F. Dusberger and G. R. Raidl. A variable neighborhood search using very large neighborhood structures for the 3-staged 2-dimensional cutting stock problem. In M. J. Blesa, C. Blum, and S. Voß, editors, *Hybrid Metaheuristics*, volume 8457 of *LNCS*, pages 85–99. Springer, 2014.

[15] F. Dusberger and G. 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*. Springer, 2015. to appear in LNCS series.

[16] F. Dusberger and G. 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.

[17] 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.

[18] K. 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.

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

[20] P. Gilmore and R. E. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations research*, 13(1):94–120, 1965.

[21] J. Herz. Recursive computational procedure for two-dimensional stock cutting. *IBM Journal of Research and Development*, 16(5):462–469, 1972.

[22] M. Hifi. An improvement of viswanathan and bagchi's exact algorithm for constrained two-dimensional cutting stock. *Computers & Operations Research*, 24(8):727–736, 1997.

[23] M. Hifi. Exact algorithms for large-scale unconstrained two and three staged cutting problems. *Computational Optimization and Applications*, 18(1):63–88, 2001.

[24] M. Hifi and R. M'Hallah. An exact algorithm for constrained two-dimensional two-staged cutting problems. *Operations Research*, 53(1):140–150, 2005.

[25] M. Hifi and V. Zissimopoulos. A recursive exact algorithm for weighted two-dimensional cutting. *European Journal of Operational Research*, 91(3):553–564, 1996.

[26] S. Hong, D. Zhang, H. C. Lau, X. Zeng, and Y.-W. Si. A hybrid heuristic algorithm for the 2d variable-sized bin packing problem. *European Journal of Operational Research*, 238(1):95–103, 2014.

[27] M. Kang and K. Yoon. An improved best-first branch-and-bound algorithm for unconstrained two-dimensional cutting problems. *International Journal of Production Research*, 49(15):4437–4455, 2011.

[28] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1):379–396, 2002.

[29] A. Lodi, S. Martello, and D. Vigo. Models and bounds for two-dimensional level packing problems. *Journal of Combinatorial Optimization*, 8(3):363–379, 2004.

[30] S. Martello and P. Toth. Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, 45(5):768–778, 1997.

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

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

[33] J. Puchinger, G. R. Raidl, and G. Koller. *Solving a real-world glass cutting problem.* Springer, 2004.

[34] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[35] S. Tschöke and N. Holthöfer. A new parallel approach to the constrained two-dimensional cutting stock problem. In *Parallel Algorithms for Irregularly Structured Problems*, pages 285–300. Springer, 1995.

[36] K. Viswanathan and A. Bagchi. Best-first search methods for constrained two-dimensional cutting stock problems. *Operations Research*, 41(4):768–776, 1993.

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

[38] L. Wei, T. Tian, W. Zhu, and A. Lim. A block-based layer building approach for the 2d guillotine strip packing problem. *European Journal of Operational Research*, 239(1):58–69, 2014.

[39] G. Young-Gun, Y.-J. Seong, and M.-K. Kang. A best-first branch and bound algorithm for unconstrained two-dimensional cutting problems. *Operations Research Letters*, 31(4):301–307, 2003.