DIPLOMARBEIT

Solving a Weighted Set Covering Problem for Improving Algorithms for Cutting Stock Problems with Setup Costs by Solution Merging

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Mathematik

eingereicht von

Dipl.-Ing. Benedikt Klocker, B.Sc.
Matrikelnummer 0926194

ausgeführt am Institut für Logic and Computation
der Fakultät für Informatik der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl

Wien, 4. April 2019

Benedikt Klocker	Günther R. Raidl

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Benedikt Klocker, B.Sc.

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, 4. April 2019

Benedikt Klocker

Acknowledgements

This thesis was developed as part of a project which was partly funded by LodeStar Technology Ges.m.b.H and the Austrian Promotion Agency FFG under contract "Innovationsscheck Plus Nr. 855569". Furthermore, part of my research was funded by the Vienna Graduate School on Computational Optimization, grant W1260.

I want to thank all for funding my research and the opportunity of working on this topic. Furthermore, I would like to show my gratitude Günther Raidl, my supervisor, for his ongoing support and advice and his constructive feedback. I would also like to expand my gratitude to all my colleagues who always had an open ear for my questions and helped me with good advice.

Furthermore, I want to thank my parents for giving me the opportunity of studying and my friends, my family and my girlfriend for continuously supporting me whenever I needed it.

Abstract

Cutting stock problems occur in many industrial applications where parts have to be cut out of raw materials. Most of the algorithms for solving such problems involve producing many cutting patterns during their execution. We propose a set cover approach which makes use of those produced patterns and searches for the best combination of a subset of them to cover all demands.

To solve the problem we propose four methods: an integer linear program (ILP) model, a greedy construction heuristic, a PILOT approach, and a beam search procedure. These algorithms are relatively independent from many specifics of the concrete variant of cutting stock problem to be solved. Furthermore, we propose a hybrid combination of the greedy set cover approach and a problem dependent construction heuristic.

The methods work especially well on problems with setup costs, which are costs for each type of used pattern. In cases with setup costs it is enough to have a problem specific algorithm which ignores pattern setup costs. The set cover approach then optimizes the solution with regard to the pattern setup costs.

We test our approaches for the K-staged two-dimensional cutting stock problem with variable sheet size and pattern setup costs. As a result of the testing, we can statistically significantly conclude that the ILP model works best on small instances, the greedy and the hybrid algorithms work best on large instances and the PILOT and beam search approach work better than the greedy and hybrid on small and medium-sized instances. Furthermore, the algorithms can improve over 50% of the tested solutions given by a problem specific algorithm if pattern setup costs are active.

Contents

CHAPTER 1

Introduction

One of the most natural tasks in many production processes in various economic areas is processing raw materials to obtain various kinds of goods. Often, such a task involves cutting out parts from the raw material in predefined sizes. Deciding how to cut the raw materials into parts such that the unusable raw material waste gets minimized is clearly an economic incentive. This task was and is in simple cases solved by humans, but especially when the number and the different types of needed parts is large it can be hard to manually find satisfactory solutions. Therefore, many companies start to use computer based solutions for calculating good cutting patterns. This lightens their employees workloads and often the algorithms find better cutting patterns than a human would ever be able to and therefore save costs.

In the literature, such problems are called cutting stock problems. They deal with cutting small items out of larger raw materials, given some specified demand for each item size. Some example industries which heavily need to solve this kind of problems are paper, film, metal, glass, and wood industries. The area of cutting stock problems is widely studied in the literature. The classical problems but also many highly specialized problem variants are considered and sophisticated solution approaches for solving them got developed.

Many of those solution approaches generate many cutting patterns during their execution. For example so called improvement heuristics try to improve a given solution by exchanging some cutting patterns by other cutting patterns. This leads naturally to a pool of patterns which may be much larger than just the patterns used in the final solution. The main idea of this thesis is to exploit this pool of potentially promising cutting patterns and find a good solution using just those cutting patterns in a postprocessing step. This idea follows the very general concept of solution merging, which uses multiple solutions for a problem and merges them together to one superior solution. For more details on the concept of solution merging see [BPRR11]. The advantage of this approach for our problem is that selecting the best combination of patterns from the pattern pool does not need to know the specific restrictions on the patterns which depend on the specific

problem variant we are considering. Therefore, the postprocessing is independent of the problem variant and the pattern restrictions, except for restrictions which relate to combinations of patterns. This implies that solution approaches for the postprocessing can be applied to a big variety of problem variants in the area of cutting stock problems.

In this thesis we will propose a general cutting stock model which covers many cutting stock problem formulations and variants. Then we will formulate a set covering problem which tries to find the best combination of patterns from a given pattern pool. Furthermore, we will propose four different approaches for solving the set covering problem and one hybrid approach which uses a set covering approach in combination with a construction approach for solving the underlying cutting stock problem.

This thesis was developed as part of a project which was financed by LodeStar Technology Ges.m.b.H. In this project an approach for solving a two-dimensional variant of the cutting stock problem was developed by Dusberger and Raidl [DR14, DR15, DR17]. The implementation of this approach was integrated in the software package by LodeStar Technology and is used by many clients from different industries, especially the wood cutting industry, to optimize their cutting patterns. A machine used for executing such cutting patterns in the wood cutting industry is shown in Figure 1.1. Because of

Figure 1.1: A cut-to-size saw for the wood cutting industry. (Image by SCHELLING Anlagenbau GmbH)

this tight relationship with the industry we have a set of real world instances for the given two-dimensional variant of a cutting stock problem and can use the algorithm by Dusberger and Raidl to generate pattern sets which can be used as instances for the set covering problem.

One of the weaknesses of the algorithm by Dusberger and Raidl is the rather weak ability to consider so called pattern setup costs. They occur in practice if a human has to adjust the cutting machine whenever it should cut a structurally different pattern. Because we want to minimize the human workload, we assign costs for changing the pattern which effectively assigns costs to each structurally different pattern. Another situation which can be modeled with pattern setup costs is when raw materials can be stacked and cut simultaneously. In this case we assign costs to the amount of time a machine needs for a cutting plan. Minimizing the time leads to a more efficient production process. Therefore, we want to have patterns with a high multiplicity so that we can stack the raw materials when cutting those patterns. In the end also leads to assigning costs to each structurally different pattern, although there may be a maximum stack size. In this case those costs have to get multiplied by the number of stacks needed for one pattern.

As already mentioned the approach by Dusberger and Raidl does not work well together with pattern setup costs. This was the main reason for us to solve this set covering problem in a postprocessing step which can then improve the solution quality by focusing on large stacks of good patterns. For testing our approaches we use the pattern sets generated by the algorithm of Dusberger and Raidl applied to different real world instances. As we will see in the test results the postprocessing is able to improve the solution for over 50% of the instances if pattern setup costs dominate the other costs. Also, with lower pattern setup costs or with no pattern setup costs the postprocessing is able to find improvements for many instances.

We also propose integrating our hybrid approach as a neighborhood into the algorithm by Dusberger and Raidl such that it is not just used as a postprocessing but regularly in the iterations of the improvement algorithm. This helps to guide the search towards solutions with good pattern setup costs and can improve the performance of the algorithm especially for instances with a strong focus on pattern setup costs.

1.1 Related Work

There is much literature in the area of cutting stock problems. Due to the high number of problem variants Dyckhoff [Dyc90] developed a typology to categorize cutting stock problems and Wäscher et al. [WHS07] extended it later on. We will look at this typology in Chapter 2 to show to what categories of cutting stock problems our set cover approach is applicable. The work by Sweeney and Paternoster [SP92] tries to summarize the research done in the field of cutting and packing problems, although this work is already quite outdated. For a survey of two-dimensional cutting stock problems see [LMM02].

There exist quite a few scientific publications about pattern setup costs in cutting stock problems. Henn and Wäscher [HW13] analyze setup costs for different cutting stock problems and consider how to adapt existing models to be able to handle setup costs. Furthermore, Belov and Scheithauer [BS07] developed an approach for solving the one dimensional cutting stock problem with setup costs.

There are already approaches in the literature which are based on the idea of reusing already generated patterns and combine them to potentially better solutions. Cui et al. [CZY15] developed a two phase approach for solving the one-dimensional cutting stock problem with pattern setup costs. The first phase generates good patterns and the second phase searches for the best solution composed out of those generated patterns which is done with an integer linear program (ILP) model. Their ILP model is similar to our model although they consider only the one dimensional case and also a slightly less general problem. For example, the problem has no availabilities of raw materials.

Another two-phase approach for the one-dimensional cutting stock problem with setup costs was proposed by Förster and Wäscher [FW00] and the second phase tries to combine patterns in a given solution into one pattern which is then used multiple times. The approach of reducing the number of different patterns in a given solution is based on the work of Diegel et al [DCVSN93].

There exist a lot of research results and many algorithms concerning the classical set covering problem [CTF00]. Our set covering problem is similar to the general weighted set covering problem, although there is no concept like setup costs for this problem. A theoretical analysis of the general weighted set covering problem was done by Yang and Leung [YL05].

The K-staged two-dimensional cutting stock problem, from which the problem we consider in our tests is derived, was already studied early by Gilmore and Gomory [GG65]. To solve this problem they introduced the general technique of column generation which is used today for many combinatorial problems in a large number of other application domains. For generating patterns for our considered variant with setup costs we use the algorithm proposed by Dusberger and Raidl [DR14, DR15, DR17].

We presented a part of the approaches and results discussed in this thesis at the sixteenth international conference on computer aided systems theory and published the work in the conference proceedings [KR18].

1.2 Structure of the Work

In the following chapter, we will explore the area of cutting stock problems and present some basic problem types. In Chapter 3 we introduce a general cutting stock problem formulation and the related cutting stock set cover problem which will be the main problem variants we consider. Chapter 4 presents four different solution approaches for solving the cutting stock set cover problem and one hybrid solution approach for solving the general cutting stock problem. The approach by Dusberger and Raidl is described in detail in Chapter 5. Chapter 6 shows the results of the approaches and evaluate the performance when adding a neighborhood based on our hybrid approach to the algorithm by Dusberger and Raidl. Finally, we conclude the thesis in Chapter 7 and present some ideas for future work.

CHAPTER 2

Cutting Stock Problems

In many economic areas a manufacturer has to cut out objects of specific sizes from raw materials, as these objects are needed in the production process. An economic incentive is to use as few raw materials as possible or equivalently to have as little waste as possible remaining after cutting the objects out. Therefore, planning how to cut the raw materials is crucial and one can consider the optimization problem of finding a good cutting plan such that the amount of used raw materials is minimal. Such optimization problems are called cutting stock problems and because of their wide variety of applications there exist many variants of them and a lot of research is done in this area.

To classify the different types and varieties of cutting and packing problems Dyckhoff [Dyc90] formulated 1990 a typology for cutting and packing problems which was later extended by Wäscher et al [WHS07]. Note that both typologies also include packing problems which can frequently be modeled in the same way as cutting problems but have other applications. Cutting problems represent the problem of cutting some parts out of bigger parts and packing problems represent packing items into bigger items, but for modeling both problems are strongly related. All problems of this type have in common that there is given a set of large objects (input, supply) and a set of small items (output, demand) have to be grouped together into possibly multiple groups to fit onto/into the large objects. Dyckhoff typology is based on four characteristics of the problems:

- The dimensionality of the objects and the items. One- and two-dimensional problems are the most common problems, although there are problem formulations for three dimensions or in general N dimensions.

- Kind of assignment: There are two possibilities. Either one has to fit all the given items onto a selection of the given objects or one has to fill all given objects with a selection of the given items.

- Assortment of large objects: There are three possibilities. Either there is only one large object, multiple objects all having the same shape, or multiple objects with different shapes
- Assortment of small items:
  - Few items with different shapes
  - Many items with many different shapes
  - Many items with few different shapes
  - All items have the same shape

The extension of Wäscher et al. distinguishes five categories:

- The dimensionality as in Dyckhoff's typology
- Kind of assignment as in Dyckhoff's typology. We will call the first case when we have to use all items and a subset of the objects *input minimization* and the second case when all objects have to be used and a subset of items *output maximization*.
- Assortment of small items:
  - identical: All items have the same shape
  - weakly heterogeneous: Items can be grouped in a small number of classes of the same shape
  - strongly heterogeneous: Only few items have same shapes
- Assortment of large objects:
  - One large object
    - The size of the object is fixed
    - The size of the object in at least one dimension is variable: Occurs most of the time with input minimization where the size of the single object has to be minimized
  - Several large objects: Here we only consider fixed size objects in contrast to the case of one large object, where we allow a single object of variable size
    - identical
    - weakly heterogeneous
    - strongly heterogeneous
- Shape of small objects:
  - regular small items: rectangles, circles, boxes, cylinders, balls, etc.
  - irregular small items

In this work we want to focus on problems dealing with input minimization as kind of assignment. In the case of input minimization using only identical small items is normally not considered since this would strongly reduce the problem and in many cases it would be trivial to solve. Therefore, we only consider the weakly heterogeneous and strongly heterogeneous type for the assortment of small items. Wäscher et al. called the problem classes with fixed kind of assignment and assortment of small items basic problem types. In our case we are interested in two basic problem types, the one with input minimization and weakly heterogeneous items, which Wäscher et al. simply called Cutting Stock Problem, and the one with input minimization and strongly heterogeneous items, which Wäscher et al. called Bin Packing Problem.

As the chapter title suggests we will focus mainly on different types of cutting stock problems but since bin packing problems are closely related to cutting stock problems we will present first the basic bin packing problem.

Bin Packing Problem. Given a bin size V and a set of items I with sizes $a_i \in \mathbb{R}_+$ for each $i \in I$. Find a minimal number of bins B and a partitioning of I into B sets $I_1, \ldots, I_B$ such that

$$\sum_{i \in I_k} a_i \leq V \quad \forall k \in \{1, \ldots, B\}.$$

This simple formulation of the bin packing problem can be classified by the classification of Wäscher et al. as follows. The dimensionality is one (the size), the kind of assignment is input minimization as we already fixed this before, the assortment of small items is strongly heterogeneous (every item has possibly another size), the assortment of large objects is several identical large objects (the bins), and the shapes of the small objects are regular (one dimensional with a size).

In the following we will focus on cutting stock problems, i.e. where we fix the assignment type to input minimization and the assortment of small items to weakly heterogeneous. The one-dimensional case has among others applications in paper, film and metal industries. In those applications large rolls of some material get produced and then have to be cut into smaller rolls with different widths, depending on the demand. Figure 2.1 shows a roll slitting machine which can cut a paper roll into smaller paper rolls.

For the two-dimensional cutting stock problems, application areas are for example the wood and glass industry. In those applications often appearing problems are of the form given a number of rectangular shaped sheets of material the question is how to cut out smaller rectangular shaped pieces with minimal waste. One could also consider non-rectangular shaped sheets or pieces, although most of the existing literature on 2-dimensional cutting stock problems considers only rectangles.

In the following we present some important standard problems.

Figure 2.1: A roll slitting machine which cuts paper rolls into smaller rolls. (Image by Soma Engineering is licensed under CC BY-SA 3.0)

2.1 One-dimensional Cutting Stock Problems

In the context of one-dimensional cutting stock problems we will call the large objects the input rods and the small items the pieces. The most basic one-dimensional problem is stated as follows.

One-dimensional cutting stock problem (1DCSP). Let T be a set of different input rod types, $L_t \in \mathbb{R}_+$ the length of the input rod of type $t \in T$ and $c_t \in \mathbb{R}_+$ the cost of the input rod of type $t \in T$. Furthermore, let E be a set of pieces and for each $i \in E$ let $l_i \in \mathbb{R}_+$ be the length of the piece and $d_i \in \mathbb{N} \setminus \{0\}$ the demand of the piece. A solution S to the problem is now a set of patterns $\mathcal{P}^S$, and for each pattern $P \in \mathcal{P}^S$ an amount a_P^S. Furthermore each pattern $P \in \mathcal{P}^S$ is associated with an input rod type $t_P \in T$. Each pattern $P \in \mathcal{P}^S$ can be represented by an element vector $(e_i^P)_{i \in E} \in \mathbb{N}^{|E|}$ where each entry of the vector is the amount of pieces of the corresponding length which get cut out of the input rod, i.e., e_i^P is the amount of pieces of length l_j used in the pattern. A solution is feasible if it covers all pieces, i.e.,

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot e_i^P = N_i \quad \forall i \in E$$

and if all patterns are valid, i.e.,

$$\sum_{i \in E} l_i \cdot e_i^P \leq L_{t_P} \quad \forall P \in \mathcal{P}^S.$$

The problem is now to find a feasible solution S which minimizes the costs of the used input rods

$$c(S) := \sum_{P \in \mathcal{P}^S} a_P^S c_{t_P}.$$

Remark 2.1.1. In the formulation of 1DCSP we implicitly assumed that the amount of available input rods of each length is unlimited.

This basic problem was one of the first problems which got introduced in the context of cutting stock problems. A classical solution approach for solving it is column generation introduced by Gilmore and Gomory [GG61]. The problem was one of the first applications of column generation and is still a classical example for the usage of column generation.

Many authors formulate classical one-dimensional cutting stock problems where all input rods have the same length. Since this is just the restriction $k = 1$ in our problem formulation, we use the more general formulation as our base problem. In the typology of Wäscher et al. [WHS07] the problem as we defined it is called one-dimensional multiple stock size cutting stock problem (1DMSSCSP) which can then be distinguished from the one-dimensional single stock size cutting stock problem (1DSSSCSP). As we will mostly operate on multiple stock sizes, we will not use this extended terminology.

There are a lot of extensions of 1DCSP many of them originating from different real world applications. In this work we focus on one special kind of extension called setup costs. Modern cutting machines often need significant time to switch between two different patterns, therefore it is encouraged to minimize the number of different patterns such that associated costs with these times get minimized. This leads to a second objective and in general therefore leads to a multi objective optimization problem. One variant which considers this situation is the one-dimensional cutting stock problem with pattern reduction (1DCSPPR) [FW00]. It only considers input rods of the same length L and searches for a solution with minimal number of used input rods and within all those solutions it searches for the solution with a minimal number of different patterns. Another approach is to use cost weights measured in a currency instead of using the number of different patterns as second level objective. This combined objective was considered among others by Mobasher and Ekici [ME13] but is not covered as well as the 1DCSPPR in the literature. Since there are different names for this problem in the literature, we will call it the one-dimensional cutting stock problem with setup costs (1DCSPSC).

We want to mention that there are many more variants of the One-Dimensional Cutting Stock Problem and what we presented in this section are just a few selected variants which are important for our approach later on.

2.2 Two-dimensional Cutting Stock Problems

For two-dimensional cutting stock problems we will call the large objects the sheets and the small items the pieces or elements. There are many categories of two-dimensional cutting stock problems, one of the most basic distinctions being regularity.

Two-dimensional regular cutting stock problem (2DRCSP). Given a sheet size (W, H) of width W and height H and a set of rectangular pieces where each piece i has a width $w_i \in \mathbb{R}_+$, a height $h_i \in \mathbb{R}_+$ and a demand $d_i \in \mathbb{N} \setminus \{0\}$. The problem is now to

distribute all the pieces (multiplied by their demand amount) inside a minimal amount of sheets of width W and height H such that no piece overlaps with another piece.

Two-dimensional irregular cutting stock problem (2DICSP). Given a sheet size (W, H) of width W and height H and a set of pieces where each piece has a demand and can be described by a (irregular) polygon without holes. The problem is again to distribute all the pieces (multiplied by their demand amount) inside a minimal amount of sheets of width W and height H such that no piece overlaps with another piece.

Note that 2DICSP is sometimes also called nesting problem in the literature. It is easy to see that 2DICSP is a generalization of 2DRCSP, although in the literature 2DRCSP is considered far more often than 2DICSP. This is probably because the regular version is easier to solve and maybe also because many problems occurring in practice naturally contain only rectangular pieces. Note that one could even further generalize 2DICSP by allowing also holes in the polygons or even allow pieces of other forms which are not polygons. Allowing pieces would increase the complexity of the problem even more since items could now also be placed within the wholes of other items. Relaxing the condition that all items are polygons would lead to problems which are hard to even formulate and even harder to solve. Furthermore, one can argue that all geometrical forms can be approximated by polygons since we allow arbitrary number of sides. One of the few works which consider the 2DICSP is the work of Albano and Sapuppo [AS80] who presented heuristics to solve the problem.

Next we will further investigate some categories of two-dimensional regular cutting stock problems. A common restriction in this class of problems is that each side of each piece is aligned parallel to one of the sides of the containing sheet. We call the 2DRCSP together with this restriction the two-dimensional orthogonal cutting stock problem (2DOCSP). Although De Cani [DC78] showed that this restriction may lead to substantially worse solutions than non-orthogonal solutions for some problem instances, most literature is only concerned with the orthogonal case of the problem.

A further restriction of the 2DOCSP which is common are so called guillotine cuts. The main idea behind this restriction is that many cutting machines can only cut through the whole material. Therefore, a *guillotine cut* is a cut (a straight line) which goes from one end of the sheet to the opposite end of the sheet. Note that it is allowed to do guillotine cuts in multiple stages, e.g. first cutting the sheet vertically and then cutting the resulting components horizontally where each component can be cut individually.

An example pattern is shown in Figure 2.2. In the first stage the red horizontal lines are cut. Then, the remaining components get cut along the blue vertical lines and last but not least the green horizontal lines get cut. Note that all the cuts always go from one side of the component to the other side and are therefore guillotine cuts. An example of a pattern which cannot be cut by guillotine cuts only is illustrated in Figure 2.3. This can easily be seen by the fact that except of the waste on the right side nothing can be cut with a guillotine cut without cutting through an output piece. We call the problem when

restricted to only guillotine cuts the two-dimensional guillotine cutting stock problem (2DGCSP).

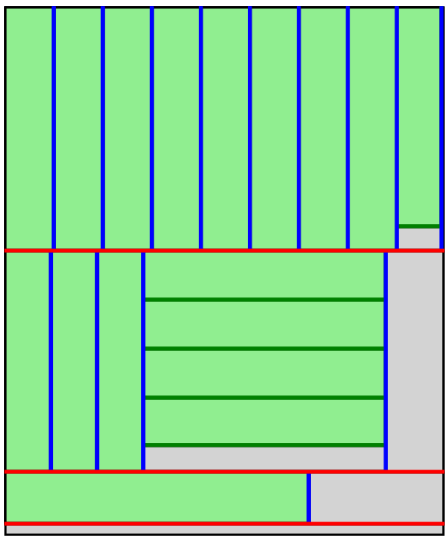

Figure 2.2: An example pattern using guillotine cuts.

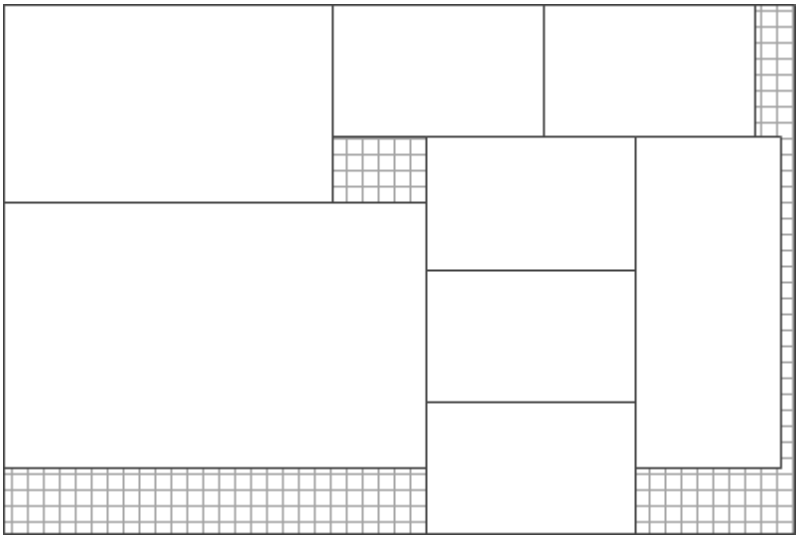

Figure 2.3: An example pattern which cannot be cut by only guillotine cuts.

If we consider only guillotine cuts, one can also consider the number of cutting stages of the patterns. For example the pattern in Figure 2.2 has three cutting stages, the red lines are cut in the first stage, the blue in the second and the green in the third. One can now restrict the number of cutting stages in the patterns to at most K stages for a $K > 0$. This problem is then called the K-staged two-dimensional cutting stock

problem (K2DCSP) [GG65]. This restriction may be inclined because of a simpler solution representation or because of practical limitations.

Up until now all two-dimensional problems presented here only use one sheet type with a given width W and height H, but many applications use sheets of different sizes in practice. Therefore, we introduce the following extension of the K2DCSP. A similar extension could be formulated for all presented two-dimensional problems until now.

K-staged two-dimensional cutting stock problem with variable sheet size (K2DCSPV). Given a set of sheet types T with widths $W_t \in \mathbb{R}_+$, heights $H_t \in \mathbb{R}_+$, available quantities $q_t \in \mathbb{N} \cup \{\infty\}$, and costs $c_t \in \mathbb{R}_+$ for $t \in T$. Furthermore, let E be a set of different element types. Each element type $i \in E$ has a width $w_i \in \mathbb{R}_+$, a height $h_i \in \mathbb{R}_+$, and a demand $d_i \in \mathbb{N} \setminus \{0\}$. A solution S to the problem is now a set of patterns $\mathcal{P}^S$ and for each pattern $P \in \mathcal{P}^S$ an amount a_P^S. Furthermore, each pattern $P \in \mathcal{P}^S$ is associated with a sheet type t_P. Each pattern describes how to cut elements out of the associated sheet type only using guillotine cuts. We can associate with each pattern $P \in \mathcal{P}^S$ an element vector $(e_i^P)_{i \in E} \in \mathbb{N}^{|E|}$ which describes how often the i-th element occurs in the pattern P. A solution is feasible if all element demands are satisfied, i.e.

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot e_i^P = d_i \quad \forall i \in E,$$

and all available sheet quantities are not exceeded, i.e.

$$\sum_{P \in \mathcal{P}^S : t_P = t} a_P^S \leq q_t \quad \forall t \in T.$$

The problem is now to find a feasible solution which minimizes the costs

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot c_{t_P}. \tag{2.1}$$

As in the one-dimensional case, we can again consider the variant of the K2DCSPV where we add pattern setup costs. As input, we get an instance of the K2DCSPV together with stacking/setup costs c_t^{S} for each sheet type t and a maximum stack size $s^{\max} \in \mathbb{N} \cup \{\infty\}$. Then the new objective is the old objective (2.1) plus the pattern setup costs

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot c_{t_P} + \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \cdot c_{t_P}^{\mathrm{S}} \tag{2.2}$$

where in the case of $s^{\max} = \infty$ the expression $\left\lceil \frac{a_P^S}{s^{\max}} \right\rceil$ is defined as 1 if $a_P^S > 0$ and as 0 otherwise.

The K-staged two-dimensional cutting stock problem with variable sheet size and pattern setup costs (K2DCSPVSC) is then defined by replacing the objecitve (2.1) by (2.2).

CHAPTER 3

The Cutting Stock Set Cover Problem

Most of the methods from the literature for solving a set cover problem, especially improvement heuristics, generate many patterns during the execution of the algorithm, more than the ones that are used in the final solution. The idea of the cutting stock set cover problem is now to make use of this variety of generated patterns and find the best solution consisting of some of those patterns. The advantages of this method are that the problem is not concerned with the concrete structure of the patterns, it only is interested in the number of items on a pattern. Therefore, all problem-specific constraints for the pattern construction do not need to be considered, except of those constraints which operate on multiple patterns or on how they can be combined.

Before we can formulate the cutting stock set cover problem we formulate a general variant of the cutting stock problem which we will use as the base problem.

General cutting stock problem (GCSP). Let E be a set of elements, $(d_i)_{i \in E} \in \mathbb{N}^{|E|}$ a demand vector and $s^{\max} \in \mathbb{N} \cup \{\infty\}$ the maximum stack size. Further, let T be a set of stock materials and $q_t \in \mathbb{N} \cup \{\infty\}$ the maximal available quantities for each stock material $t \in T$.

A solution is represented by a multiset of patterns. A pattern is in general a structural collection of elements in E which satisfies some problem-specific constraints. The structure of the pattern is also problem-specific, but irrelevant for this approach. We can associate with each pattern P an element vector $(e_i^P)_{i \in E} \in \mathbb{N}^{|E|}$ which describes how often the element i is contained in the pattern P. Furthermore, a pattern P is associated to a stock material $t_P \in T$ out of which it gets cut. A pattern P has associated problem specific production costs $c_P^{\mathrm{P}} \geq 0$ and stacking costs $c_P^{\mathrm{S}} \geq 0$. Note that the set of possible patterns, its production costs and stacking costs are not part of the problem instance, since depending on which problem the possible patterns are described by problem-specific

structural rules and constraints and the costs are described as formulas possibly depending on the structure of the pattern and the used stock materials and elements.

We define a solution S as a set of feasible patterns $\mathcal{P}^S$ and an amounts vector $(a_P^S)_{P\in\mathcal{P}^S} \in \mathbb{N}^{|\mathcal{P}^S|}$. The goal is to find an optimal solution S which satisfies the demand constraints

$$\sum_{P\in\mathcal{P}^S} e_i^P \cdot a_P^S \geq d_i \quad \forall i \in E, \tag{3.1}$$

the stock material availability constraints

$$\sum_{P\in\mathcal{P}^S:t_P=t} a_P^S \leq q_t \quad \forall t \in T, \tag{3.2}$$

and minimizes the total costs

$$c(s) := \sum_{P\in\mathcal{P}^S} c_P^{\mathrm{P}} \cdot a_P^S + \sum_{P\in\mathcal{P}^S} \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \cdot c_P^{\mathrm{S}}. \tag{3.3}$$

If $s^{\max} = \infty$ we define $\left\lceil \frac{a_P^S}{s^{\max}} \right\rceil$ equals 1 if $a_P^S > 0$ and 0 if $a_P^S = 0$.

We further consider the problem variant *general cutting stock problem with exact demands (GCSPE)* in which demands must exactly be satisfied, which means we replace condition (3.1) by

$$\sum_{P\in\mathcal{P}^S} e_i^P \cdot a_P^S = d_i \quad \forall i \in E. \tag{3.4}$$

If we compare the GCSP with the typology of Wäscher et al., see chapter 2, we see that many problem types fit into the framework of GCSP. The only restrictions are that the kind of assignment is input minimization and that the dimensions of the large objects (the stock material) are fixed. All problems presented in chapter 2 fit into the framework. Moreover, many problem variants and extensions deal with special constraints for the pattern construction, especially for problem formulations closer to real world situations. All those restrictions on the pattern structure can be considered in the GCSP since it makes no assumptions on the problem-specific constraints for the pattern structure.

Note that the classical cutting stock problems do not allow overproduction as it is allowed in GCSP in contrast to GCSPE. If we do not have any setup costs overproduction makes no sense since you could simply remove the overproduced elements without increasing the costs, this is why all classical problems do not consider overproduction. But, as soon as we introduce pattern setup costs overproduction makes sense, since it may be useful to overproduce some elements to be able to stack the patterns together. Removing the overproduced elements would lead to more stacks and therefore higher costs.

Since the costs are non-negative, we can assume that patterns in an optimal solution are not empty. In fact in most applications patterns always have positive costs, so an

optimal solution never has empty patterns. This assumption implies that the maximal amount of stock materials used is limited by $D := \sum_{i \in E} d_i$. Therefore instead of using $s^{\max} = \infty$ we can always equivalently use $s^{\max} = D$. In the same way we can use $q_t = D$ instead of $q_t = \infty$. To simplify the algorithms presented in this thesis we will assume from now on that $s^{\max} \in \mathbb{N}$ and $q_t \in \mathbb{N}$ for all $t \in T$.

Using the GCSP we can now formulate the cutting stock set cover problem.

Cutting stock set cover problem (CSSCP). Let E, $(d_i)_{i \in E}$, $s^{\max}$, T and q_t for $t \in T$ be given as in the GCSP. Furthermore, let $\mathcal{P}$ be a given finite set of feasible patterns (e.g. collected from different heuristic solutions). The CSSCP asks for a solution S to the underlying GCSP consisting of patterns in $\mathcal{P}$, i.e. $\mathcal{P}^S \subseteq \mathcal{P}$ which satisfies the conditions (3.1) and (3.2) and minimizes the costs $c(S)$ as defined in (3.3).

If we replace condition (3.1) by (3.4), we call the problem *cutting stock set cover problem with exact demands (CSSCPE).*

CHAPTER 4

Solution Approaches

In this chapter we present four different approaches for solving the CSSCP, which are an integer linear program for solving the problem exactly, a greedy approach for finding reasonable solutions fast, a PILOT-approach, and a beam search.

4.1 Integer Linear Programming Formulation

We start by modeling the CSSCP as integer linear program (ILP). Theoretically it can solve the problem exactly, but in practice the approach does not scale well to large instances. Therefore, if we use a time limit it may produce solutions with large optimality gaps. We use integer variables a_P for the amount of each pattern P and integer variables s_P for the number of stacks of the pattern P.

$$\begin{aligned}
\min_{(a_P)_{P\in\mathcal{P}},(s_P)_{P\in\mathcal{P}}} & \sum_{P\in\mathcal{P}} a_P \cdot c_P^{\mathrm{P}} + s_P \cdot c_P^{\mathrm{S}} \\
\text{s.t.} \quad & \sum_{P\in\mathcal{P}} a_P \cdot e_i^P \geq d_i && \forall i \in E && (4.1)\\
& \sum_{P\in\mathcal{P}:t_P=t} a_P \leq q_t && \forall t \in T && (4.2)\\
& s_P \cdot s^{\max} \geq a_P && \forall P \in \mathcal{P} && (4.3)\\
& a_P \in \mathbb{N}, s_P \in \mathbb{N} && \forall P \in \mathcal{P}
\end{aligned}$$

If we want to solve CSSCPE, we replace constraint (4.1) by

$$\sum_{P\in\mathcal{P}} a_P \cdot e_i^P = d_i \quad \forall i \in E. \tag{4.4}$$

The constraints (4.1) or (4.4) ensure that the demands get satisfied and the inequalities (4.2) guarantee that the maximal amounts for each stock material get respected. Furthermore, the constraints (4.3) couple the s_P variables with the a_P variables by ensuring that there are enough stacks, so that the maximal stack size $s^{\max}$ gets not exceeded.

4.2 Greedy Heuristic

Since the ILP model can only compute solutions for small instances in reasonable time, we need other approaches for larger instances. The idea of the following greedy construction heuristic is to rate each pattern depending on the current unsatisfied demands and pick the best pattern as the next one greedily.

We greedily add patterns to a partial solution until the demands are all satisfied. Because of the greedy nature and the restricted pattern possibilities of the CSSCP/CSSCPE, it is easier to design a greedy heuristic for the CSSCP than the CSSCPE. Therefore, we will present in the following a greedy construction heuristic for the CSSCP and will talk afterwards how we could modify it to solve the CSSCPE.

Before we can describe the algorithm, we formalize what a partial solution is.

Definition 4.2.1 (Partial Solution of CSSCP, CSSCPE, GCSP, and GCSPE)**.** Let E, $(d_i)_{i\in E}$, $s^{\max}$, T and q_t for $t \in T$ be an instance of the GCSP. A partial solution S for this instance is a multiset of feasible patterns, described by $\mathcal{P}^S$ together with an amounts vector $(a_P^S)_{P\in\mathcal{P}^S}$ which satisfies (3.2) but not necessarily (3.1). Such an S is a partial solution of the GCSPE if additionally

$$\sum_{P\in\mathcal{P}^S} e_i^P \cdot a_P^S \leq d_i \quad \forall i \in E \tag{4.5}$$

holds.

If we are further given a set $\mathcal{P}$ of feasible patterns forming an instance of CSSCP, a partial solution for that instance is a partial solution of the underlying GCSP which only consists of patterns in $\mathcal{P}$. Furthermore, for the CSSCPE a partial solution must additionally satisfy (4.5).

Note that the empty set is always a partial solution of all four problems. Therefore, the greedy heuristic starts with a valid partial solution $S = \emptyset$ of CSSCP, and adds patterns to it until the partial solution is a feasible solution, i.e. it satisfies (3.1).

We give now in each iteration based on the current partial solution S each pattern a rating and then add the pattern with the highest rating to S. To also consider stacking costs we allow to add a pattern with a given amount at once. Thus, we do not only pick a pattern but also an amount for this pattern greedily. Therefore, we need to define a rating for each pair $(P, a) \in \mathcal{P} \times \mathbb{N}$ consisting of a pattern and an amount.

For this rating we will use a problem-depending volume value $v_i \in \mathbb{R}_+$ which represents the volume of element i or in general how difficult it is to add the element i to some pattern. In most applications we can use a volume (length for one-dimensional, area for two-dimensional, volume for three-dimensional, ...), but in theory one could use any values here, even values which dynamically change during the execution of the algorithm.

Using this volume value v_i we can define the rating of a pattern and an amount by the sum of all volume values of all elements on the pattern whose demand is not yet satisfied divided by the cost of the pattern and the pattern stack. Formally we define, based on a partial solution S, the rating $r^S(P,a)$ for the pair $(P,a) \in \mathcal{P} \times \mathbb{N}$ by

$$r^S(P,a) := \frac{\sum_{i \in E} \min(a \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left(\left\lceil \frac{a + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \right)} \tag{4.6}$$

where the remaining demand r_i^S of element i is defined by

$$r_i^S := \max\left(0, d_i - \sum_{P \in \mathcal{P}} e_i^P \cdot a_P^S\right).$$

It might happen that pattern P is already used in S, i.e. $a_P^S > 0$. Therefore, we may be able to add some amount of patterns to S without creating a new stack. Thus, to calculate the amount of new stacks introduced, we use the difference of the stacks for pattern P used in the old solution with the stacks for pattern P used in the new solution after adding a times the pattern P.

In the case where $c_P^{\mathrm{P}} = 0$ it may happen that the rating is not well-defined since the denominator may be 0. In this case we have no costs for adding that amount of patterns. Therefore, we can add a maximal amount of those patterns such that we still have no costs without comparing to any other ratings. From now on assume therefore that the denominator is always larger than zero.

We also have to consider the remaining amount of the stock material and therefore we are only interested in amounts $a \in A_P^S := \{n \in \mathbb{N} : n \leq R_{t_P}^S\}$ where the remaining amount of stock material R_t^S is defined by

$$R_t^S := q_t - \sum_{P \in \mathcal{P} : t_P = t} a_P^S.$$

What we want now to find is a pair $(P_0, a_0) \in \mathcal{P} \times \mathbb{N}$ with $a_0 \in A_{P_0}^S$ whose rating $r^S(P_0, a_0)$ is maximal, i.e.

$$r^S(P_0, a_0) = \max_{(P,a) \in \mathcal{P} \times \mathbb{N} : a_0 \in A_{P_0}^S} r^S(P,a).$$

If one of the stock materials have an infinite available quantity q_t then A_P^S will be all natural numbers for all patterns using this material t. In this case there are infinitely

many possible pairs (P, a) and therefore we need to only consider relevant ones to find the maximum. The following theorems establish properties of the rating function and help us to develop an efficient algorithm for finding an optimal amount a for a pattern P.

Theorem 4.2.1. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$ together with $\mathcal{P}$ be an instance of CSSCP and S a partial solution of this instance. Furthermore, let $P \in \mathcal{P}$ be a pattern with $A_P^S \neq \emptyset$ and r be the remaining amount of the pattern P in a not yet finished stack in S, i.e. $0 \leq r < s^{\max}$ and $r \equiv a_P^S \mod s^{\max}$. Then it holds*

$$r^S(P, s^{\max} - r) \geq r^S(P, a) \quad \forall a \geq s^{\max} - r.$$

Proof. We distinguish the cases $r = 0$ and $r > 0$. If $r = 0$ we have

$$\begin{aligned}
r^S(P,a) &= \frac{\sum_{i\in E} \min(a \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left\lceil \frac{a}{s^{\max}} \right\rceil} \leq \frac{\sum_{i\in E} \min(a \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \frac{a}{s^{\max}}} \\
&= \frac{\sum_{i\in E} \min(e_i^P, \frac{r_i^S}{a}) \cdot v_i}{c_P^{\mathrm{P}} + \frac{c_P^{\mathrm{S}}}{s^{\max}}} \leq \frac{\sum_{i\in E} \min(e_i^P, \frac{r_i^S}{s^{\max}}) \cdot v_i}{c_P^{\mathrm{P}} + \frac{c_P^{\mathrm{S}}}{s^{\max}}} \\
&= \frac{\sum_{i\in E} \min(s^{\max} \cdot e_i^P, r_i^S) \cdot v_i}{s^{\max} \cdot c_P^{\mathrm{P}} + c_P^{\mathrm{S}} \frac{s^{\max}.}{s^{\max}}} = \frac{\sum_{i\in E} \min(s^{\max} \cdot e_i^P, r_i^S) \cdot v_i}{s^{\max} \cdot c_P^{\mathrm{P}} + c_P^{\mathrm{S}} \left\lceil \frac{s^{\max}.}{s^{\max}} \right\rceil}. \\
&= r^S(P, s^{\max}) = r^S(P, s^{\max} - r).
\end{aligned}$$

For $r > 0$ it holds

$$\left\lceil \frac{s^{\max} - r + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil = 0$$

and therefore we can calculate

$$\begin{aligned}
r^S(P,a) &= \frac{\sum_{i\in E} \min(a \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left(\left\lceil \frac{a + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \right)} \leq \frac{\sum_{i\in E} \min(a \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot a} \\
&= \frac{\sum_{i\in E} \min(e_i^P, \frac{r_i^S}{a}) \cdot v_i}{c_P^{\mathrm{P}}} \leq \frac{\sum_{i\in E} \min(e_i^P, \frac{r_i^S}{s^{\max} - r}) \cdot v_i}{c_P^{\mathrm{P}} \cdot a} \\
&= \frac{\sum_{i\in E} \min((s^{\max} - r) \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot (s^{\max} - r)} \\
&= \frac{\sum_{i\in E} \min((s^{\max} - r) \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot (s^{\max} - r) + c_P^{\mathrm{S}} \left(\left\lceil \frac{s^{\max} - r + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \right)} = r^S\left(P, s^{\max} - r\right).
\end{aligned}$$

□

The previous theorem showed us that it is enough to search for amounts smaller or equal to $s^{\max} - r$ to find one with a maximum rating. In the next steps we want to analyze the development of the sequence $(r_a)_{a=1}^{a^{\max}}$ where $r_a := r^S(P, a)$ for $a \in A_P^S$ and $a^{\max} := \min\left(s^{\max} - r, R_{t_P}^S\right)$.

Lemma 4.2.2. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$, $\mathcal{P}$, S, P, and r be as in Theorem 4.2.1. Furthermore, let $1 \le a < a^{\max}$ be fixed. Using the three element index sets*

$$\begin{aligned} I_1^a &:= \{i \in E : ae_i^P > r_i^S\}, \\ I_2^a &:= \{i \in E : ae_i^P \le r_i^S < (a+1)e_i^P\}, \\ I_3^a &:= \{i \in E : (a+1)e_i^P \le r_i^S\} \end{aligned}$$

it holds for each relation $\sim \in \{<, \le, =, \ge, >\}$ that $r_{a+1} \sim r_a$ if and only if $0 \sim S^a$ with

$$\begin{aligned} S^a :=& \sum_{i\in I_1^a} v_i x_{i,1} + \sum_{i\in I_2^a} v_i x_{i,2}^a + \sum_{i\in I_3^a} v_i x_{i,3}, \\ x_{i,1} :=& r_i^S c_P^P && \forall i = 1,\dots,n, \\ x_{i,2}^a :=& \left(c_P^P a + c_P^S \mathbb{1}_{r=0} + c_P^P\right) ae_i^P - \left(c_P^P a + c_P^S \mathbb{1}_{r=0}\right) r_i^S && \forall i = 1,\dots,n, \\ x_{i,3} :=& -e_i^P c_P^S \mathbb{1}_{r=0} && \forall i = 1,\dots,n. \end{aligned}$$

Proof. Because $a \le a+1 \le a^{\max} \le s^{\max} - r$, we get that

$$\left\lceil \frac{a + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil = \mathbb{1}_{r=0} \quad \text{and} \quad \left\lceil \frac{a+1+a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil = \mathbb{1}_{r=0}.$$

With that we can calculate

$$\begin{aligned} r_{a+1} \sim r_a \Leftrightarrow & \quad \frac{\sum_{i\in E} \min\left((a+1)e_i^P, r_i^S\right) v_i}{c_P^{\mathrm{P}}(a+1) + c_P^{\mathrm{S}} \mathbb{1}_{r=0}} \sim \frac{\sum_{i\in E} \min\left(ae_i^P, r_i^S\right) v_i}{c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}} \\ \Leftrightarrow & \quad \left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right) \sum_{i\in E} \min\left((a+1)e_i^P, r_i^S\right) v_i \\ & \quad \sim \left(c_P^{\mathrm{P}}(a+1) + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right) \sum_{i\in E} \min\left(ae_i^P, r_i^S\right) v_i \\ \Leftrightarrow & \quad 0 \sim \sum_{i\in E} v_i \Big(\left(c_P^{\mathrm{P}}(a+1) + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right) \min\left(ae_i^P, r_i^S\right) - \\ & \qquad \left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right) \min\left((a+1)e_i^P, r_i^S\right) \Big) \\ \Leftrightarrow & \quad 0 \sim \sum_{i\in I_1^a} v_i r_i^S c_P^{\mathrm{P}} + \\ & \qquad \sum_{i\in I_2^a} v_i \left(\left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0} + c_P^{\mathrm{P}}\right) ae_i^P - \left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right) r_i^S\right) + \\ & \qquad \sum_{i\in I_3^a} v_i e_i^P \left(\left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0} + c_P^{\mathrm{P}}\right) a - \left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right)(a+1)\right). \end{aligned}$$

The fact that

$$\left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0} + c_P^{\mathrm{P}}\right) a - \left(c_P^{\mathrm{P}} a + c_P^{\mathrm{S}} \mathbb{1}_{r=0}\right)(a+1) = c_P^{\mathrm{P}} a - c_P^{\mathrm{P}} a - c_P^{\mathrm{S}} \mathbb{1}_{r=0} = -c_P^{\mathrm{S}} \mathbb{1}_{r=0}$$

completes the proof. □

Theorem 4.2.3. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$, $\mathcal{P}$, S, P, and r be as in Theorem 4.2.1. Then there exists an amount $1 \leq a_0 \leq a^{\max}$ such that the sequence $r_1, \dots, r_{a_0}$ is monotonically increasing and the sequence $r_{a_0}, \dots, r_{a^{\max}}$ is monotonically decreasing. Furthermore, if $r > 0$ we can choose $a_0 = 1$.*

Proof. Let a_0 be the first element in $[1, a^{\max})$ for which $r_{a_0} > r_{a_0+1}$. If no such a_0 exists we can choose $a_0 = a^{\max}$ and are done. We prove now that $r_{a+1} \leq r_a$ holds for all $a_0 \leq a < a^{\max}$ by induction on a.

Induction basis $a = a_0$: $r_{a_0+1} \leq r_{a_0}$ is true by definition of a_0.

Induction step $a \to a+1$: We consider the equivalent condition $0 \leq S^{a+1}$ of Lemma 4.2.2. Note that $x_{i,3} \leq x^a_{i,2} \leq x_{i,1}$ for all i regardless of the value a. Furthermore, we have $I_1^{a+1} = I_1^a \cup I_2^a$, and $I_3^a = I_2^{a+1} \cup I_3^{a+1}$. Therefore, we get

$$\begin{aligned} S^{a+1} &= \sum_{i\in I_1^{a+1}} v_i x_{i,1} + \sum_{i\in I_2^{a+1}} v_i x^{a+1}_{i,2} + \sum_{i\in I_3^{a+1}} v_i x_{i,3} \\ &\geq \sum_{i\in I_1^{a}} v_i x_{i,1} + \sum_{i\in I_2^{a}} v_i x_{i,1} + \sum_{i\in I_2^{a+1}} v_i x_{i,3} + \sum_{i\in I_3^{a+1}} v_i x_{i,3} \\ &\geq \sum_{i\in I_1^{a}} v_i x_{i,1} + \sum_{i\in I_2^{a}} v_i x^a_{i,2} + \sum_{i\in I_3^{a}} v_i x_{i,3} = S^a \geq 0 \end{aligned}$$

The last inequality follows from the induction hypothesis with Lemma 4.2.2 and therefore we get all in all $0 \leq S^{a+1}$ which is equivalent to $r_{a+2} \leq r_{a+1}$.

Therefore, we finished proving the monotonicity properties, what remains to prove is that we can choose $a_0 = 1$ if $r > 0$ which is the case if the values $r_1, \dots, r_{a^{\max}}$ are monotonically decreasing. Note that in the case of $r > 0$ we have $\mathbb{1}_{r=0} = 0$ and therefore $x_{i,3} = 0 \leq x^a_{i,2} \leq x_{i,1}$ for all elements i and amounts $a < a^{\max}$. This implies that all terms of the sum S^a are non-negative for all $a < a^{\max}$ and therefore $0 \leq S^a$ for all $a < a^{\max}$. Using now Lemma 4.2.2 gives us that $r_{a+1} \leq r_a$ for all $a < a^{\max}$ which is what we wanted to show. □

Theorem 4.2.4. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$, $\mathcal{P}$, S, P, and r be as in Theorem 4.2.1. Then, there exists an amount a_0 in*

$$A_P^{S,0} := \{a^{\max}\} \cup \left\{ \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor, \left\lceil \frac{r_i^S}{e_i^P} \right\rceil : i \in E, e_i^P > 0 \right\} \cap \{a \in A_P^S : a \leq a^{\max}\}$$

which has a maximum rating, i.e. $r^S(P, a_0) \geq r^S(P, a)$ for all $a \in A_P^S$, and it is the largest of all amounts that have a maximum rating and are smaller or equal to $a^{\max}$.

Proof. Let a_0 be the first element in $[1, a^{\max})$ for which $r_{a_0} > r_{a_0+1}$. If such an a_0 does not exist, then $a_0 := a^{\max}$ satisfies all properties. Note that in this case $a^{\max}$ must have a maximum rating because of Theorem 4.2.1.

In the other case if an $a_0 < a^{\max}$ with $r_{a_0} > r_{a_0+1}$ exists we know b9 Theorem 4.2.3 together with Theorem 4.2.1 that a_0 must have a maximum rating and that it is the largest of all amounts with this property which are smaller or equal to $a^{\max}$.

The only thing remaining to show is that $a_0 \in A_P^{S,0}$. If $a_0 = 1$, then we know by Lemma 4.2.2 that either I_1^1 or I_2^1 are not empty since $r_2 < r_1$ implies that $0 < S^1$. If I_1^1 is not empty, then there exists an element i with $e_i^P > r_i^S$ which implies

$$a_0 = 1 = \left\lceil \frac{r_i^S}{e_i^P} \right\rceil \in A_P^{S,0}.$$

On the other hand if I_2^1 is not empty then there exists an element i with $e_i^P \leq r_i^S < 2e_i^P$ which implies

$$a_0 = 1 = \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor \in A_P^{S,0}.$$

The only remaining case is now if $a_0 > 1$. In this case we know that $r_{a_0-1} \leq r_{a_0}$ and $r_{a_0+1} < r_{a_0}$ and therefore, we get that $S^{a_0-1} \leq 0 < S^{a_0}$ which implies $S^{a_0} \neq S^{a_0-1}$. Assume now there would not exist an i with $(a-1)e_i^p < r_i^S < (a_0+1)e_i^p$. This would imply that $I_2^{a_0}$ is empty and that there may only be elements $i \in I_2^{a_0-1}$ with $(a_0-1)e_i^P = r_i^S$. But for those i we get $x_{i,1} = x_{i,2}^{a_0-1}$ and we get

$$\begin{aligned} S^{a_0-1} &= \sum_{i \in I_1^{a_0-1}} v_i x_{i,1} + \sum_{i \in I_2^{a_0-1}} v_i x_{i,1} + \sum_{i \in I_3^{a_0-1}} v_i x_{i,3} = \sum_{i \in I_1^{a_0}} v_i x_{i,1} + \sum_{i \in I_3^{a_0}} v_i x_{i,3} \\ &= S^{a_0} \end{aligned}$$

which is a contradiction. Therefore, there exists an i with $(a_0-1)e_i^p < r_i^S < (a_0+1)e_i^p$. If $r_i^S \leq a_0 e_i^p$ we get that

$$a_0 = \left\lceil \frac{r_i^S}{e_i^p} \right\rceil \in A_P^{S,0}$$

and otherwise if $r_i^S > a_0 e_i^p$ we get that

$$a_0 = \left\lfloor \frac{r_i^S}{e_i^p} \right\rfloor \in A_P^{S,0}.$$

□

Theorem 4.2.5. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$, $\mathcal{P}$, S, P, and r, $A_P^{S,0}$ be as in Theorem 4.2.4. Furthermore, let a_0 be the largest amount in A_P^S which has a maximum rating, i.e. $r^S(P, a_0) \geq r^S(P, a)$ for all $a \in A_P^S$. If $r > 0$ and $c_P^S > 0$ then it holds $a_0 \in A_P^{S,0}$. If $r = 0$ and $c_P^S > 0$ then it holds that either $a_0 \in A_P^{S,0}$ or $a_0 = k \cdot s^{\max}$ for some $k \in \mathbb{N} \setminus \{0\}$ and $j \cdot s^{\max}$ has a maximum rating for all $1 \leq j \leq k$.*

Proof. We consider the proof of Theorem 4.2.1. If $r > 0$ and $c_P^{\mathrm{S}} > 0$ then $r^S(P, a_0) = r^S(P, s^{\max} - r)$ can only be true if

$$\left\lceil \frac{a_0 + a_P^S}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil = 0 \Leftrightarrow a_0 \leq s^{\max} - r.$$

Therefore, $a_0 \leq a^{\max}$ which implies by Theorem 4.2.3 that $a_0 \in A_P^{S,0}$. On the other hand if $r = 0$ and $c_P^{\mathrm{S}} > 0$ then $r^S(P, a_0) = r^S(P, s^{\max})$ can only be true if

$$\left\lceil \frac{a_0}{s^{\max}} \right\rceil = \frac{a_0}{s^{\max}} \Leftrightarrow \exists k \in \mathbb{N} : a_0 = k \cdot s^{\max}.$$

For $1 \leq j \leq k$ we get

$$r^S(P, j \cdot s^{\max}) = \frac{\sum_{i \in E} \min(j \cdot s^{\max} \cdot e_i^P, r_i^S) \cdot v_i}{c_P^{\mathrm{P}} \cdot j \cdot s^{\max} + j \cdot c_P^{\mathrm{S}}} = \frac{\sum_{i \in E} \min(s^{\max} \cdot e_i^P, \frac{r_i^S}{j}) \cdot v_i}{c_P^{\mathrm{P}} \cdot s^{\max} + c_P^{\mathrm{S}}}$$
$$\geq \frac{\sum_{i \in E} \min(s^{\max} \cdot e_i^P, \frac{r_i^S}{k}) \cdot v_i}{c_P^{\mathrm{P}} \cdot s^{\max} + c_P^{\mathrm{S}}} = r^S(P, k \cdot s^{\max})$$

which implies that $j \cdot s^{\max}$ has a maximum rating for all $1 \leq j \leq k$. □

Using Theorems 4.2.1–4.2.5 we can verify that Algorithm 1 correctly computes for a given pattern P the largest amount $a_0 \in A_P^S$ with a maximum rating. The running time of Algorithm 1 is in $\mathcal{O}(m^2)$ where $m \leq n$ is the amount of item types on the input pattern P, i.e. all $i \in E$ with $e_i^P > 0$. Note that line 18 of Algorithm 1 can be implemented efficiently by iterating through the different relevant item types and check for which k they get saturated. Therefore, the bottlenecks of the algorithm are the lines 16 and 18 which both need $\mathcal{O}(m^2)$ time since computing $r^S(P, a)$ needs $\mathcal{O}(m)$ time and $|A_P^{S,0}| \leq m$.

The whole greedy approach is described in Algorithm 2.

Lemma 4.2.6. *Let $a, b \in \mathbb{N}$, $0 < a \leq b$, then*

$$a \left\lfloor \frac{b}{a} \right\rfloor > \frac{b}{2}.$$

Proof. Since $a > 0$, the statement is equivalent to

$$\left\lfloor \frac{b}{a} \right\rfloor > \frac{1}{2} \cdot \frac{b}{a}.$$

Since $a \leq b$, we know $\left\lfloor \frac{b}{a} \right\rfloor \geq 1$. If $\frac{b}{a} < 2$ we are therefore done. On the other hand if $\frac{b}{a} \geq 2$ then we can calculate

$$\left\lfloor \frac{b}{a} \right\rfloor > \frac{b}{a} - 1 \geq \frac{b}{a} - \frac{1}{2} \cdot \frac{b}{a} = \frac{1}{2} \cdot \frac{b}{a}.$$

□

Algorithm 1 FindBestA(S, P)

INPUT: An instance of CSSCP, a partial solution S, and a pattern P
OUTPUT: The largest $a_0 \in A_P^S$ with a maximum rating

if $c_P^{\mathrm{S}} = 0$ **then**
  $a_0 \leftarrow \min_{i \in E: e_i^P > 0 \wedge r_i^S > 0} \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor$
  if $a_0 = 0$ **then**
    $a_0 \leftarrow 1$ ▷ Already the first pattern overproduces some elements
  end if
  if $a_0 > R_{t_P}^S$ **then**
    $a_0 \leftarrow R_{t_P}^S$
  end if
  return a_0
else
  Compute r as in Theorem 4.2.1.
  if $r > 0$ and $c_P^{\mathrm{P}} = 0$ **then**
    return $a_0 := a^{\max}$ ▷ No costs for filling the stack
  end if
  Compute the set $A_P^{S,0}$
  Find first a_0 in $A_P^{S,0}$ where either $a_0 = a^{\max}$ or $r^S(P, a_0) > r^S(P, a_0 + 1)$
  if $r = 0$ and $a_0 = s^{\max}$ **then**
    Find first $k \in \mathbb{N}$ with $(k+1)s^{\max} > R_{t_P}^S$ or $r^S(P, s^{\max}) > r^S(P, (k+1)s^{\max})$
    $a_0 \leftarrow k s^{\max}$
  end if
  return a_0
end if

Theorem 4.2.7. *The worst case run time of Algorithm 2 is*

$$\mathcal{O}\left(\left[n\Big(\log(\max_{i \in E} d_i) + 1\Big) + |T|\Big(\log(\max_{t \in T} q_t) + 1\Big)\right] \cdot |\mathcal{P}| \cdot n^2\right)$$

Proof. If we can prove that the number of iterations of the while loop is in

$$\mathcal{O}\left(n\Big(\log(\max_{i \in E} d_i) + 1\Big) + |T|\Big(\log(\max_{t \in T} q_t) + 1\Big)\right),$$

we are done since FindBestA is in $\mathcal{O}(n^2)$. Next we prove the following statement. In each iteration of the while loop, when we add pattern P_{best} with amount a_{best} then at least one of the following happens:

1. There exists an element i for which the remaining demand is non-zero and gets more than halved.

Algorithm 2 Greedy Construction Heuristic

INPUT: An instance of CSSCP
OUTPUT: A solution to the CSSCP or INFEASIBLE if no solution could be found

```
1: Set S to the empty partial solution
2: while S does not satisfy (3.1) do
3:     r_best ← −1
4:     for P ∈ 𝒫 do
5:         if R^S_{t_P} > 0 then
6:             a ← FindBestA(S, P)
7:             if r^S(P, a) > r_best then
8:                 r_best ← r^S(P, a)
9:                 (P_best, a_best) ← (P, a)
10:            end if
11:        end if
12:    end for
13:    if r_best = −1 then
14:        return INFEASIBLE
15:    else
16:        add pattern P_best with amount a_best to the partial solution S
17:    end if
18: end while
19: return S
```

2. There exists a material type $t \in T$ for which the remaining available amount is non-zero and gets more than halved.

3. A previously not completely filled stack in S gets completely filled.

There are six different cases for where the return value gets set the last time in Algorithm 1 which computes a_{best}:

1. Line 2: In this case the remaining demand of item i for which the minimum is achieved is reduced by

$$e_i^P \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor$$

which is greater than $\frac{r_i^S}{2}$ by Lemma 4.2.6 and the fact that since we know that $a_0 > 0$ it must hold $e_i^P < r_i^S$. Therefore, the remaining demand of item i got more than halved.

2. Line 4: In this case the remaining demand of item i for which the minimum in line 2 was achieved is set to 0 which is also more than halved.

3. Line 7: In this case the remaining available amount of type t_P is set to 0 and therefore more than halved.

4. Line 13: A previously not completely filled stack got completely filled or all the remaining amount of material type t_P is used.

5. Line 16: Either $a_0 = a^{\max}$, in which case the same happens as in the previous case, or $a_0 = \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor$ for some element type i, in which case the remaining demand of i gets more than halved, or $a_0 = \left\lceil \frac{r_i^S}{e_i^P} \right\rceil$ in which case the remaining demand of i gets set to 0.

6. Line 19: If $(k+1)s^{\max} > R_{t_P}^S$ we know that the remaining available amount of type t_P gets more than halved. On the other hand if $r^S(P, s^{\max}) > r^S(P, (k+1)s^{\max})$ this implies that at least one element type i was not overproduced by adding the pattern $ks^{\max}$ times but gets overproduced by adding it $(k+1)s^{\max}$ times. Therefore, we know $r_i^S \geq ks^{\max}e_i^P$ and $r_i^S < (k+1)s^{\max}e_i^P$ which implies that

$$ks^{\max}e_i^P \geq \frac{k+1}{2}s^{\max}e_i^P > \frac{r_i^S}{2}.$$

   Therefore, the remaining demand of item type i gets more than halved.

A natural number n can only get more than halved no more than $\log_2(n)+1$ many times before it hits 0. To see this let a_k be the natural number after k steps of more than halving. Since $a_0 = n$, we get $a_k < \frac{1}{2}^k n$. If $a_k \geq 1$ this implies $1 < \frac{1}{2}^k n$ which implies $n > 2^k$ and therefore $k < \log_2(n)$. Therefore, for $k \geq \log_2(n)$ the value of a_k must be 0.

Using this fact we see that case 1 can only happen in at most $n(\log_2(\max_{i\in E} d_i) + 1)$ iterations and case 2 can only happen in at most $|T|(\log_2(\max_{t\in T} q_t) + 1)$ iterations. Since case 3 can only happen in at most half of the iterations, we are done. □

Note that the running time proven in Theorem 4.2.7 is only the worst case running time and in most cases the running time is much closer to $|P|n^3$.

To further reduce the running time we can use the following fact:

Theorem 4.2.8. *Let E, $(d_i)_{i\in E}$, $s^{\max}$, T, and q_t for $t \in T$ together with $\mathcal{P}$ be an instance of CSSCP. Furthermore, let S_j be the partial solution of iteration j when running Algorithm 2 on this instance.*

Then it holds that

$$\max_{a\in A_P^{S_j}} r^{S_j}(P,a) \geq \max_{a\in A_P^{S_k}} r^{S_k}(P,a) \quad \forall (P,a) \in \mathcal{P}\times\mathbb{N}, j \leq k.$$

To prove Theorem 4.2.8 we use the following Lemma.

Lemma 4.2.9. *Let $a, b, c, d \in \mathbb{R}_+ \cup \{0\}$ and $b \neq 0, c \neq 0$. Then it holds*

$$\frac{a}{b} \geq \frac{a+c}{b+d} \Leftrightarrow \frac{a}{b} \geq \frac{c}{d}$$

Proof. It simply follows by the following basic equivalent transformation:

$$\frac{a}{b} \geq \frac{a+c}{b+d} \Leftrightarrow ab + ad \geq ab + bc \Leftrightarrow ab \geq bc \Leftrightarrow \frac{a}{b} \geq \frac{c}{d}.$$

□

Proof of Theorem 4.2.8. It is enough to show the statement for $k = j + 1$ since then everything else follows by the transitivity of the inequality. We fix j and $k = j + 1$. Since we always only add patterns to a partial solution in Algorithm 2, we know that $a_P^{S_k} \geq a_P^{S_j}$ for all $P \in \mathcal{P}$. From that it directly follows

$$r_i^{S_j} \geq r_i^{S_k}.$$

Furthermore, it follows $R_{t_P}^{S_j} \geq R_{t_P}^{S_k}$ and therefore $A_P^{S_j} \supseteq A_P^{S_k}$. Let $\tilde{P}$ be the pattern added to S_k in iteration j, i.e. S_k consists of all patterns in S_j plus some amount of the pattern $\tilde{P}$.

We distinguish now the cases $\tilde{P} \neq P$ and $\tilde{P} = P$. In the first case we know that $a_P^{S_j} = a_P^{S_k}$ and therefore

$$\begin{aligned}
r^{S_j}(P, a) &= \frac{\sum_{i \in E} \min(a \cdot e_i^P, r_i^{S_j}) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left(\left\lceil \frac{a + a_P^{S_j}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{S_j}}{s^{\max}} \right\rceil \right)} \\
&= \frac{\sum_{i \in E} \min(a \cdot e_i^P, r_i^{S_j}) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left(\left\lceil \frac{a + a_P^{S_k}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{S_k}}{s^{\max}} \right\rceil \right)} \\
&\geq \frac{\sum_{i \in E} \min(a \cdot e_i^P, r_i^{S_k}) \cdot v_i}{c_P^{\mathrm{P}} \cdot a + c_P^{\mathrm{S}} \left(\left\lceil \frac{a + a_P^{S_k}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{S_k}}{s^{\max}} \right\rceil \right)} = r^{S_k}(P, a) \quad \forall a \in A_P^{s_j} \supseteq A_P^{s_k}.
\end{aligned}$$

Therefore, the remaining case is when $\tilde{P} = P$. Let a_j be the amount of pattern $\tilde{P}$ which got added to S_j in iteration j. That implies that a_j has a maximum rating for S_j. Furthermore, let a_k be an amount with maximum rating for S_k. Then we know since a_j was maximal for S_j that $r^{S_j}(P, a_j) \geq r^{S_j}(P, a_j + a_k)$. From that we get

$$\begin{aligned}
r^{s_j}(P, a_j) \geq r^{s_j}(P, a_j + a_k) &= \frac{\sum_{i \in E} \min((a_j + a_k) \cdot e_i^P, r_i^{s_j}) \cdot v_i}{c_P^{\mathrm{P}} \cdot (a_j + a_k) + c_P^{\mathrm{S}} \left(\left\lceil \frac{a_j + a_k + a_P^{s_j}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{s_j}}{s^{\max}} \right\rceil \right)} \\
&= \frac{\sum_{i \in E} \min(a_j \cdot e_i^P, r_i^{s_j}) \cdot v_i + \sum_{i \in E} \min(a_k \cdot e_i^P, r_i^{s_k}) \cdot v_i}{c_P^{\mathrm{P}} \cdot a_j + c_P^{\mathrm{S}} \left(\left\lceil \frac{a_j + a_P^{s_j}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{s_j}}{s^{\max}} \right\rceil \right) + c_P^{\mathrm{P}} \cdot a_k + c_P^{\mathrm{S}} \left(\left\lceil \frac{a_k + a_P^{s_k}}{s^{\max}} \right\rceil - \left\lceil \frac{a_P^{s_k}}{s^{\max}} \right\rceil \right)}
\end{aligned}$$

We can apply now Lemma 4.2.9 and get that $r^{s_j}(P, a_j) \geq r^{s_k}(P, a_k)$. □

Theorem 4.2.8 basically tells us that we can use the calculated ratings from previous iterations as upper bounds for the current rating of a pattern. Therefore, we store the latest calculated rating r_P^{latest} of each pattern. Then when we iterate through all patterns in P in an iteration we start with the patterns with the highest rating r_P^{latest}. Whenever we calculate a new rating we replace r_P^{latest} for this pattern. If the currently best rating of the patterns already calculated in the current iteration is higher than r_P^{latest} for some pattern P we do not have to recalculate that pattern since it will always be worse than the currently best pattern. Therefore, in each iteration we only have to calculate some of the ratings for the best patterns and in many practical cases this reduces the running time drastically, although the worst-case running time complexity stays the same.

4.2.1 Greedy Approach with Exact Demands

In this section we will present how to modify the presented greedy approach to be able to solve CSSCPE, which uses exact demands.

The naive way to modify the greedy approach is to modify 1 such that the found amount a is always such that no items get overproduced. That means that we change $a^{\max}$ to

$$a^{\max} := \min\left(s^{\max} - r, R^S_{t_P}, \min_{i \in E: e_i^P > 0} \left\lfloor \frac{r_i^S}{e_i^P} \right\rfloor\right).$$

Although this would give us a correct algorithm for CSSCPE, in many practical cases and for many instances it would not be able to find a feasible solution. To improve solution qualities for exact demands we therefore formulate a new problem variant of the cutting stock set cover problem.

Cutting stock sub set cover problem for exact demands (CSSSCPE). Let E, $(d_i)_{i \in E}$, $s^{\max}$, T and q_t for $t \in T$ be given as in the GCSP. Furthermore, let P be a given finite set of feasible patterns (e.g. collected from different heuristic solutions). The CSSSCPE asks for a solution to the underlying GCSPE consisting of patterns derived from patterns in $\mathcal{P}$. By derived, we mean that for each pattern P_0 in the solution there exists a pattern $P \in \mathcal{P}$ such that $e_i^{P_0} \leq e_i^P$, i.e. P_0 can be derived from P by deleting some elements from the pattern.

We can modify now our greedy algorithm in a simple way to efficiently solve the CSSSCPE. The only thing we have to do is whenever we add a pattern in line 16 of Algorithm 2 we remove unneeded elements from the patterns. Note that we also need to change the calculation of the amount a since we only can increase the amount a until before the next element on the pattern gets overproduced. So we can use the same calculation but only allow values in $A_P^{S,0}$ which do not overproduce any elements with still open demands (the elements which have no open demand get removed from the pattern anyway and are therefore not relevant for the stack calculation).

Note that the solving approaches for CSSCP were completely independent of the problem specific constraints on patterns. But for the CSSSCPE we may now also consider some problem specific constraints to not be violated when removing elements from a pattern. Therefore, it may happen that removing some elements from the best pattern P will fail in the greedy approach. In that case we will search for the second best pattern P_2 according to the greedy criterion. If removing unneeded elements from P_2 also fails we search for the third best pattern and so on until either we found a pattern for which the removal works or we stop the greedy algorithm with an infeasible partial solution.

4.3 PILOT-Approach

The preferred iterative look ahead technique (PILOT) is a well studied method to improve the performance of a construction heuristic. The method was proposed by Duin and Voß [DV99]. Its idea is to use an embedded simpler heuristic to complete a partial solution and use the objective value from the completed solution to rate the partial solution.

In the classical PILOT approach in each iteration the partial solution obtained after adding every possible extension is completed by a given construction heuristic. Then the extension corresponding to the best obtained complete solution gets applied. In Algorithm 3 a generic classical PILOT approach is presented.

Algorithm 3 Classical PILOT

INPUT: An instance of a discrete optimization problem
OUTPUT: A solution to the problem

1: Set the partial solution $S \leftarrow \emptyset$ to the empty solution.
2: **while** S is not complete **do**
3: **for** e possible extension of S **do**
4: $S' \leftarrow S \cup \{e\}$ ▷ the partial solution after applying the extension e to S
5: complete S' with a construction heuristic and remember rating r_e
6: **end for**
7: Apply the extension e with the best rating r_e to S, i.e. $S \leftarrow S \cup \{e\}$
8: **end while**

To apply the classical PILOT approach for our problems CSSCP and CSSSCPE we use as construction heuristic the greedy heuristic presented in Section 4.2. Furthermore, as extensions of a partial solution we consider for each pattern $P \in \mathcal{P}$ only the pair (P, a) with $a = \text{FindBestA}(S, P)$. That means we do not consider every pair (P, a) as an extension, but for each pattern only one pair with the largest amount a which has a maximum greedy rating for this pattern.

It is easy to see that the worst case running time of the classical PILOT is the maximum number of extensions which have to be applied until a solution is complete times the

maximum number of extensions of a partial solution times the worst case running time of the construction heuristic. In our case this leads to a worst case running time of

$$\mathcal{O}\left(\left[n\Big(\log(\max_{i\in E} d_i)+1\Big)+|T|\Big(\log(\max_{t\in T} q_t)+1\Big)\right]^2\cdot|\mathcal{P}|^2\cdot n^2\right).$$

Although, as already mentioned in Section 4.2, the worst case running time is not reached in many practical applications the running time of the classical PILOT is often too slow for large scale real-world instances. Therefore, we also consider a parameterized version of PILOT which restricts the possible extensions in each iteration to the best β extensions by the greedy criterion. Note that this restriction holds only for the PILOT outer loop, not for the extension ratings within the greedy construction heuristic.

The modified version of the PILOT applied to CSSCP is described in Algorithm 4. The

Algorithm 4 Restricted PILOT with Parameter β

INPUT: An instance of a discrete optimization problem
OUTPUT: A solution to the problem

1: set the partial solution $S \leftarrow \emptyset$ to the empty solution
2: **while** S is not complete **do**
3: compute for all patterns $P \in \mathcal{P}$ the amount $a_P \leftarrow \text{FindBestA}(S, P)$
4: compute for all patterns $P \in \mathcal{P}$ the rating $r_P \leftarrow r^S(P, a_P)$
5: set $\mathcal{P}_0$ to the β best patterns according to r_P
6: **for** $P \in \mathcal{P}_0$ **do**
7: copy S to S' and add pattern P with amount a_P to S'
8: complete S' with Algorithm 2
9: store objective of complete solution o_P
10: **end for**
11: set P_0 to the pattern with the best corresponding objective o_P
12: add P_0 with amount a_{P_0} to S
13: **end while**

worst-case running time of the restricted version is now

$$\mathcal{O}\left(\left[n\Big(\log(\max_{i\in E} d_i)+1\Big)+|T|\Big(\log(\max_{t\in T} q_t)+1\Big)\right]^2\cdot\beta\cdot|\mathcal{P}|\cdot n^2\right)$$

which makes a big difference in practical applications, depending on β.

To solve CSSSCPE we can modify the PILOT approach in a similar way as we did for the greedy approach in Section 4.2.1. First of all we replace Algorithm 2 by the greedy algorithm which solves CSSSCPE as described in Section 4.2.1. Furthermore, on line 7 we need to remove overproduced elements from the pattern before we add it to S'. If that fails, we skip the pattern P and add the next best pattern to P_0, beginning with the $k+1$-th pattern, if one more exists. We also need to remove overproduced elements before we add the pattern P_0 to S in line 12. If that fails, we consider for P_0 the next best pattern in P_0 according to the objective values o_P.

4.4 Beam Search Approach

Beam Search is well-known extension of greedy construction heuristics with the goal to improve the exploration of the search space [Low76, MCF+77]. Instead of only storing the best solution and following the best extensions it stores the best β solutions for a parameter β.

Algorithm 5 Beam Search

INPUT: An instance of a discrete optimization problem
OUTPUT: A solution to the problem

1: Initialize the set of current partial solutions $\mathcal{S} \leftarrow \emptyset$
2: Add the empty partial solution $S = \emptyset$ to $\mathcal{S}$
3: **while** $\mathcal{S}$ is not empty **do**
4: **for** $S \in \mathcal{S}$ **do**
5: **for** e possible extension of S **do** Compute a rating $r(S')$ for $S' \leftarrow S \cup \{e\}$
6: **end for**
7: **end for**
8: Clear $\mathcal{S}$
9: Store the β best partial solutions S', according to $r(S')$, in $\mathcal{S}$
10: If any S' is complete store the best objective o^{best} and its solution
11: Remove all partial solutions from $\mathcal{S}$ which cannot be better than o^{best}
12: **end while**
13: **return** the best found complete solution

In the beam search approach in each iteration it checks all possible extensions of all stored solutions, applies them and keeps the best β partial solutions in the memory. The whole beam search approach is presented in Algorithm 5 in a generic manner.

To apply Algorithm 5 to our problem we have to decide how we rate partial solutions. We propose to do that similarly as we rated extensions, volume of all satisfied demands divided through the costs. For a partial solution S' we define the rating $r(S')$ by

$$r(S') := \frac{\sum_{i \in E} \max\left(d_i, \sum_{P \in \mathcal{P}} e_i^P \cdot a_P^{S'}\right) \cdot v_i}{\sum_{P \in \mathcal{P}} c_P^{\mathrm{P}} \cdot a_P^{S'} + c_P^{\mathrm{S}} \cdot \left\lceil \frac{a_P^{S'}}{s^{\max}} \right\rceil}.$$

Furthermore, as extensions we use the same as we did for the PILOT approach. That means for each pattern P there is one extension consisting of adding the pattern with the largest amount that has a maximum rating $a_P = \mathrm{FindBestA}(S, P)$. Finally, to check if a partial solution can still be better than the best objective found until now, we just have to compare the costs, since the costs may only get worse to complete the partial solution.

Using the beam search as formulated above for our problem leads to unsatisfactory solution qualities compared to the time spent in the beam search. The problem is that

since we compute for each pattern different amounts a, and also since different patterns may be of completely different sizes and costs (depending on the used raw material), it happens that some partial solutions have already much more item demands satisfied than other partial solutions in the same stage. Since the rating of the patterns always decreases when more patterns get added, see Theorem 4.2.8, the solutions which satisfy more demands have in average lower ratings than the solutions which satisfy fewer demands (and also have fewer costs). This leads to a bias of the rating function for partial solutions which satisfy few demands but also have few costs, compared to solutions which satisfy more demands and have higher costs. This is intuitively also clear since the fewer demands are left, the harder it gets to find a good pattern which satisfies exactly those demands. Furthermore, also stacking gets more difficult the fewer demands are left.

But this bias is exactly what we do not want since the solutions which have high stacks, i.e. they get filled in a few iterations, since each iteration adds a high stack of patterns, are potentially the best solutions. Therefore, we propose a variant of the beam search approach which distinguishes partial solutions by levels. One can also think of this variant in the sense that normal beam search is a form of tree search where in each iteration we operate on the best k nodes on the same tree level. In a normal tree an edge connects a node from level n to level $n+1$, but in our case we want to allow edges in the tree which have different lengths. That means an edge does not always connect a node from level n with a node from level $n+1$ but in general a node from level n with a node from level $n+k$. We then say the length of the edge is k.

For our problem we define the level of a partial solution S by

$$l(s) := \left\lfloor \frac{c(s)}{c^{\text{unit}}} \right\rfloor$$

where $c(s)$ are the costs of the partial solution S and $c^{\text{unit}} > 0$ is the cost granularity of the tree. That means what we do is to choose a cost granularity and then group partial solutions into buckets based on their costs. In our case we propose to use the smallest pattern cost $c^{\text{unit}} := c_P^{\text{P}}$. In the edge case that there are patterns with no costs we could also consider the stacking costs divided through the maximum stack of those patterns. If there is a pattern which has no costs and no stacking costs, then we just use the costs of the first pattern which has some costs, but this does not occur in any practical situations.

Algorithm 6 describes the whole beam search approach with levels for the CSSCP. Note that since we have a minimization problem and solutions are grouped by their objective values it is enough to stop the beam search as soon as we found the first complete solution and finished going through the bucket of this solution, since the best solution we can find is of course in the first bucket, which contains a complete solution. If the solutions in a bucket are sorted by cost, we can even stop the algorithm as soon as the first complete solution is encountered.

The running time of the beam search approach heavily depends on the number of patterns in the solutions since this is in the worst case the number of times the outer while loop is

Algorithm 6 Set Cover Beam Search Approach

```
1: Add empty solution to storage of level 0
2: l ← 0
3: F ← ∅
4: while |F| = ∅ and partial solution for a level larger or equal l exists do
5:     for S in storage of level l do
6:         if S is complete then
7:             F ← F ∪ {S}
8:         else
9:             for P ∈ P do
10:                compute best amount a_P ← FindBestA(S, P) for pattern P
11:                create copy S' of S and add P with amount a to S'
12:                if r(S') is one of the best β ratings in level l(S') then
13:                    add S' to storage of level l(S')
14:                    if storage of level l(S') has more than β solutions then
15:                        Remove the worst solution from storage of level l(S')
16:                    end if
17:                end if
18:            end for
19:        end if
20:    end for
21:    increase l
22: end while
23: return best solution in F
```

iterated. Although in general the number of patterns in the found solutions are much less, a worst case upper bound for the number of patterns in a partial solution is the sum of all element demands

$$D := \sum_{i \in E} d_i.$$

Using that we get that the worst case running time of the beam search algorithm is

$$\mathcal{O}(D \cdot \beta \cdot |\mathcal{P}| \cdot n^2).$$

For solving problem CSSSCPE we can modify the beam search algorithm in a similar way as we did for the Greedy and the PILOT approach. When we add a pattern to a solution copy in line 11, we remove overproduced items from the pattern. If the removal fails we skip the pattern P and continue with the next pattern in $\mathcal{P}$.

4.5 Hybridizing With a Construction Heuristic

One of the main drawbacks of constructing a solution for problem CSSCP is that at the end when the partial solution is almost complete and only few demands are left it is

unlikely that there is a suitable pattern in $\mathcal{P}$ which suits the remaining demand. Therefore, the longer the construction algorithm runs the poorer the quality of the best pattern in $\mathcal{P}$ gets, as was also shown by Theorem 4.2.8. To counterfeit this drawback we can hybridize a construction heuristic for CSSCP with a construction heuristic for GCSP. Ideally at the beginning the construction heuristic for CSSCP finds good patterns which can be stacked frequently and then when the demand decreases at some point the construction heuristic for GCSP takes over and constructs new patterns which directly suit the remaining demand.

Clearly this approach does not solve the problem CSSCP since the construction heuristic for GCSP will construct patterns which are not in $\mathcal{P}$. But the solution produced by this approach is still a valid solution for the GCSP. Therefore, the approach presented in this section is a construction approach to find a promising solution for the GCSP under the assumption that we already found a lot of good patterns $\mathcal{P}$ through other methods.

To formalize the approach we assume that we already are given the set $\mathcal{P}$ of patterns and additionally we are given a construction method constructPattern which constructs one new pattern according to the remaining demand. That means the method gets as input an instance of GCSP and a partial solution S of GCSP and returns a new pattern P which may not be in $\mathcal{P}$ anymore.

Using this construction method we can formalize a hybrid approach which is presented in Algorithm 7.

Note that the only difference to the greedy algorithm are the lines 3 and 4. In each iteration of the hybrid a new pattern gets generated with constructPattern and added to the pattern set $\mathcal{P}$. Since this pattern is directly constructed for the remaining demands, it will be the best pattern in $\mathcal{P}$ if all other patterns do not suite the remaining demands well and at that point the generated patterns will be used instead of the given patterns at the beginning of the algorithm.

The worst-case running time of the hybrid approach depends on the worst case running time of constructPattern, if that is in $\mathcal{O}(g(I))$ for some function $g(I))$ we get for the whole hybrid approach a worst case running time of

$$\mathcal{O}\left(\left[n\Big(\log(\max_{i\in E} d_i)+1\Big)+|T|\Big(\log(\max_{t\in T} q_t)+1\Big)\right]\cdot\left(g(I)+|\mathcal{P}|\cdot n^2\right)\right).$$

That implies that in the case that if constructPattern is in $\mathcal{O}(|P|\cdot n^2)$ then the worst case running time of the hybrid approach is the same as for the greedy approach. For practical purposes having a fast construction method implies that the hybrid needs almost the same time as the greedy, but gives us better solution qualities.

To solve the problem with exact demands, i.e. GCSPE, we can use the same modifications as presented for the greedy in Section 4.2.1. This even works if constructPattern does not consider exact demands, but of course if it also considers exact demands the patterns produced by it will be much more useful in the hybrid and therefore it will increase the solution quality of the hybrid.

Algorithm 7 Hybrid Greedy Construction Heuristic

INPUT: An instance I of CSSCP
OUTPUT: A solution to the GCSP or INFEASIBLE if no solution could be found

Set S to the empty partial solution
while S does not satisfy (3.1) **do**
    $P_0 \leftarrow$ constructPattern(I, S)
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_0\}$
    $r_{\text{best}} \leftarrow -1$
    for $P \in \mathcal{P}$ **do**
        if $R^S_{t_P} > 0$ **then**
            $a \leftarrow$ FindBestA(S, P)
            if $r^S(P, a) > r_{\text{best}}$ **then**
                $r_{\text{best}} \leftarrow r^S(P, a)$
                $(P_{\text{best}}, a_{\text{best}}) \leftarrow (P, a)$
            end if
        end if
    end for
    if $r_{\text{best}} = -1$ **then**
        return INFEASIBLE
    else
        add pattern P_{best} with amount a_{best} to the partial solution S
    end if
end while
return S

CHAPTER 5

Solving the K-staged Two-Dimensional Cutting Stock Problem with Variable Sheet Size

In this chapter we will present an approach for solving the K-staged two-dimensional cutting stock problem with variable sheet size, which was developed by Dusberger and Raidl [DR14, DR15, DR17]. We will use this approach to generate instances of CSSCP and CSSSCPE for which we then test our approaches. Furthermore, we present how to incorporate our hybrid construction heuristic, Algorithm 7, as a new neighborhood into the solving procedure of Dusberger and Raidl and test how this improves the algorithm.

As a reminder we repeat in the following the problem formulation of the K2DCSPV as presented in Chapter 2.

K-staged two-dimensional cutting stock problem with variable sheet size (K2DCSPV). Given a set of sheet types T with widths $W_t \in \mathbb{R}_+$, heights $H_t \in \mathbb{R}_+$, available quantities $q_t \in \mathbb{N} \cup \{\infty\}$, and costs $c_t \in \mathbb{R}_+$ for $t \in T$. Furthermore, let E be a set of different element types. Each element type $i \in E$ has a width $w_i \in \mathbb{R}_+$, a height $h_i \in \mathbb{R}_+$, and a demand $d_i \in \mathbb{N} \setminus \{0\}$. A solution S to the problem is now a set of patterns $\mathcal{P}^S$ and for each pattern $P \in \mathcal{P}^S$ an amount a_P^S. Furthermore, each pattern $P \in \mathcal{P}^S$ is associated with a sheet type t_P. Each pattern describes how to cut output elements out of the associated sheet type only using guillotine cuts. We can associate with each pattern $P \in \mathcal{P}^S$ an element vector $(e_i^P)_{i \in E} \in \mathbb{N}^{|E|}$ which describes how often the i-th element occurs in the pattern P. A solution is feasible if all element demands are satisfied, i.e.

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot e_i^P = d_i \quad \forall i \in E,$$

and all available sheet quantities are not exceeded, i.e.

$$\sum_{P \in \mathcal{P}^S : t_P = t} a_P^S \leq q_t \quad \forall t \in T.$$

The problem is now to find a feasible solution which minimizes the costs

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot c_{t_P}. \tag{5.1}$$

The approach by Dusberger and Raidl is a variable neighborhood search (VNS) metaheuristic combined with very large neighborhood search (VLNS) techniques. Therefore, before we present the approach we will shortly introduce what VNS and VLNS is. Then we present the approach by Dusberger and Raidl.

5.1 Variable Neighborhood Search

Variable neighborhood search (VNS) is a common metaheuristic used for solving many problems in literature [HM99, HM03]. The approach is an improvement heuristic in contrast to the approach techniques we used in Chapter 2. That means that the approach needs an already feasible solution to start with and then tries to improve this solution iteratively. Its main idea is to use multiple neighborhood structures to escape local optima of a single neighborhood structure. By a neighborhood structure we formally mean a function which maps a solution of the search space to a set of neighbors of the solution, i.e. a subset of the search space. Often neighborhoods are represented by a move operator which describes how to change a solution to get to its neighbors.

Example 5.1.1 (Traveling salesperson problem (TSP))**.** Consider the following famous problem. Given a complete undirected graph G on n vertices together with edge weights w_{ij}. Find a tour which visits each vertex exactly once and returns to the starting vertex, i.e. a Hamiltonian cycle, in G which has a minimal tour weight.

For this problem many neighborhood structures have been proposed in the literature. A simple move for the TSP would be moving one city visit to another place in the tour. This gives us a neighborhood structure, where the neighbors of one tour are all tours which only differ by one city which is placed at another position. Other moves are the so called k-exchanges, which remove k edges and add again k edges in such a way that the result is again a tour which is different from the original tour. For each such k we get a neighborhood structure which is called k-exchange neighborhood.

If we are given now neighborhood structures $N_1, \ldots, N_k$ we can iterate through the neighborhood structures and search through each of the neighborhoods of the current solution if there exists a neighbor with a better objective value. If we find such a neighbor we assign the neighbor as the current solution and restart the procedure, starting again with neighborhood structure N_1. When we searched at some point through

all neighborhoods and could not find a better neighbor we know that the current solution is a local optimum according to all the neighborhood structures, and we return the solution. This procedure is called variable neighborhood descend (VND).

One can extend a VND-procedure to a VNS-procedure by using additionally so called shaking neighborhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_\ell$. Those additional neighborhood structures are normally much larger than the neighborhood structures $N_1, \ldots, N_k$ used for the VND and are not used for searching through them. Instead, they are used to randomly select a neighbor and jump to this neighbor. This is done whenever the VND could not improve anymore and then the VND is applied again to the new solution. If there are multiple shaking neighborhood structures the procedure uses the next neighborhood structure for shaking whenever the shaking move plus the succeeding VND could not improve the current solution and whenever the solution could get improved it starts again with the first shaking neighborhood structure. Therefore, the shaking part is from the structure again a VND wrapped around the VND with the difference that moves are applied randomly instead of searching through the neighborhoods systematically. This whole procedure is also called general variable neighborhood search (GVNS).

A simplification of the GVNS is the so called reduced variable neighborhood search (RVNS), which only uses shaking neighborhoods without applying any VND after the shaking. So it only selects a random neighbor, if it is better applies it and restarts with the first neighborhood structure and if its not better it uses the next neighborhood structure. Algorithm 8 shows how a RVNS works in detail.

Algorithm 8 Reduced variable neighborhood search (RVNS)

INPUT: An instance of a given optimization problem, a feasible starting solution S
OUTPUT: A possibly improved solution

```
1:  i ← 1
2:  while termination criterion is not satisfied do
3:      Select random neighbor S' of S in N_i
4:      if S' has a better objective than S then
5:          S ← S'
6:          i ← 1
7:      else if i = ℓ then
8:          i ← 1
9:      else
10:         i ← i + 1
11:     end if
12: end while
13: return S
```

The approach by Dusberger and Raidl which we will present later on in this chapter can be seen as a RVNS which uses very large neighborhoods as neighborhood struc-

tures. Although, in this approach moves are not generated completely random but use construction heuristics.

5.2 Very Large Neighborhood Search

The idea of very large neighborhood search is to have a large neighborhood, for which it is not feasible to enumerate all neighbors, but for which exists an efficient method to find the best neighbor or at least heuristically find a good neighbor [AEOP02, PR10].

There are many techniques and approaches for searching through large neighborhoods efficiently. Often exact approaches using network flows, dynamic programming, or similar techniques are used. Another kind of large neighborhoods are variable-depth neighborhoods and again another kind are ruin-and-recreate based neighborhoods. Since the large neighborhoods used by the approach of Dusberger and Raidl are all ruin-and-recreate based, we will go further into detail of those kinds of large neighborhoods.

A move in a ruin-and-recreate neighborhood is described by applying a ruin method and then applying a recreate method. The ruin method randomly removes parts of a solution or unassigns variables, depending on the structure of the solution. Then, given the resulting partial solution the recreate method applies a construction heuristic to repair the solution. If the removed parts of the solution were not optimal, the construction heuristic may find an improved solution in which case the search continuous from the improved solution. If the solution could not get improved another ruin and recreate move is applied until we find one which improves the solution or a termination criterion is satisfied.

Example 5.2.1. Continuing example 5.1.1 for the TSP we can define a ruin method by removing a random sub-path from the tour. The length of the sub-path is randomly selected out of a predefined range and then the first removed city is randomly selected. To repair now the partial tour we can apply for example the best fit insertion heuristic which inserts cities at the position in the partial tour where they fit best.

If we consider not just one large neighborhood but many ruin-and-recreate based large neighborhoods, we can use a variable neighborhood search technique to use them all in one algorithm. Given ruin-and-recreate based large neighborhoods $\mathcal{N}_1, \ldots, \mathcal{N}_k$ we can use them in a RVNS framework as described in Algorithm 8 in Section 5.1. It is important to note that ruin-and-recreate based large neighborhoods always generate random moves which fits into the framework of RVNS.

5.3 Solution Representation

Before we describe the algorithmic details for solving the K2DCSPV we specify how a solution gets represented. In this section we describe how a solution gets represented in the approach by Dusberger and Raidl [DR14].

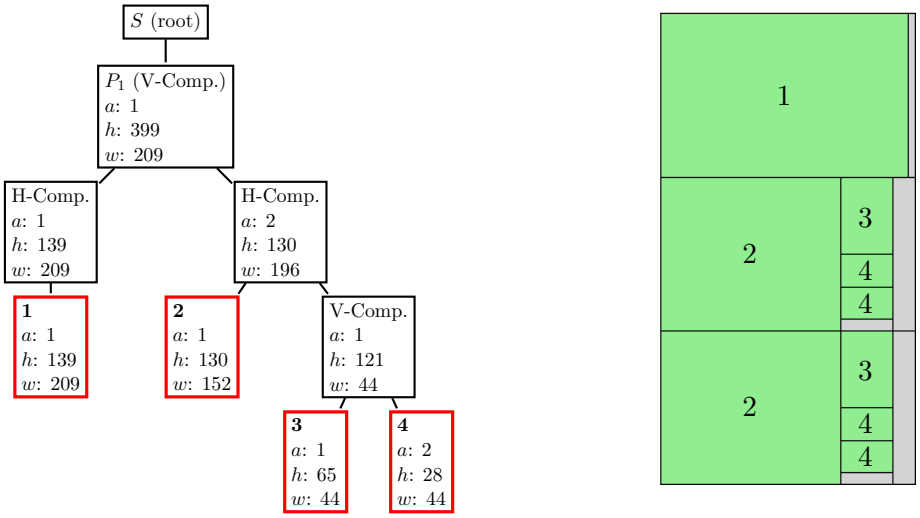

Figure 5.1: The pattern tree and a visualization of an example solution S consisting of one pattern P_1.

One important restriction which changes the nature of the problem a lot is the restriction of only allowing guillotine cuts. Every guillotine cut cuts a pattern into two halves. This fact can be used to represent a pattern naturally as a binary tree, where each inner node represents one cut. Having a discrete representation of a pattern like a tree helps a lot to deal with solutions. Furthermore, cuts which only cut off some waste from one element do not need to be part of the cutting tree. With that simplification we can enforce that the leafs of the cutting tree can each be associated with an element. Moreover, we do not need to store any positions of the cuts. Since every leaf node represents an element, we can recursively calculate where we have to place cuts starting from the leaf nodes.

We can transform the binary tree into a tree by storing parallel cuts on the same level. That implies that each level of the tree either only contains vertical cuts or horizontal cuts.

Formally a solution of the K2DCSPV is represented by only one tree, where the children of the root node represent the root nodes of the cutting trees for the respective pattern. There are three types of inner nodes, the root node of the whole tree, vertical compounds and horizontal compounds. Leaf nodes are always associated with an element type. All nodes except the root node, regardless if inner node or leaf node, store a given amount $a \in \mathbb{N} \setminus \{0\}$. This helps a lot to scale the solution representation and also the algorithms which work on the representation, since duplicate patterns or subpatterns get stored only once with a higher amount. Therefore, an algorithm can improve for example a subpattern and this gets automatically applied to all copies of this subpattern. See Figure 5.1 for an example of a cutting tree for a solution which consists of only one pattern.

As already mentioned we do not need to specify where the cuts have to be placed, since the cutting tree already induces a minimal cutting position for each cut. To calculate this cutting position we calculate and store for each node except for the root node its width w and its height h. Whenever the tree gets modified, we need to adjust those values.

The width and the height of a leaf node is simply the width w_i and the height h_i of the associated element type i. Furthermore, for a horizontal component C^{h} with children $C_1, \ldots, C_\ell$ the width can be calculated by

$$w(C^{\mathrm{h}}) := \sum_{i=1}^{\ell} w(C_i) \cdot a(C_i)$$

where $w(C_i)$ is the width of child C_i and $a(C_i)$ the amount of child C_i. The height of C^{h} is simply the maximum of all heights of the children $C_1, \ldots, C_\ell$.

For a vertical component C^{v} with children $C_1, \ldots, C_\ell$ the calculations are analogue, the height is calculated by

$$h(C^{\mathrm{v}}) := \sum_{i=1}^{\ell} h(C_i) \cdot a(C_i)$$

and the width is the maximum of the widths of the children. Based on the widths or heights of a components children one can easily calculate where to place the real cuts.

5.3.1 Waste Rectangles

Additional to the width and height we can define so called waste rectangles for all compounds of a cutting tree. A waste rectangle of a compound basically describes the largest rectangle which could get added as a child to the compound such that the sheet of the compound (the level 1 ancestor node of the compound node) still fits into the sheet type used for this sheet.

Formally, we first define a maximum width $w_P^{\max}$, a maximum height $h_P^{\max}$, a slack width $\tilde{w}_P$, and a slack height $\tilde{h}_P$ for each pattern P (all nodes, except the root node) in the cutting tree. Note that each pattern P with width w_P and height h_P is part of a sheet which is associated with a sheet type $t \in T$ with a height H_t and a width W_t. The maximum width and height of a pattern describes the maximum size of the pattern such that the containing sheet pattern would still fit into its sheet type of size (H_t, W_t). The slack height and slack width of a pattern P is simply the difference of the maximum width and height to the actual width and height, i.e.

$$(\tilde{h}_P, \tilde{w}_P) := (h_P^{\max} - h_P, w_P^{\max} - w_P).$$

We define the maximum height $h_P^{\max}$ and maximum width $w_P^{\max}$ recursively depending on type of the containing parent compound C and its slack height $\tilde{h}_C$ and slack width $\tilde{w}_C$:

$$(h_P^{\max}, w_P^{\max}) := \begin{cases} (H_t, W_t), & \text{if } P \text{ is a level 1 node, i.e. a sheet pattern.} \\ (h_C^{\max}, w_P + \tilde{w}_C), & \text{if } P \text{ is a subpattern of hor. compound } C. \\ (h_P + \tilde{h}_C, w_C^{\max}), & \text{if } P \text{ is a subpattern of vert. compound } C. \end{cases} \tag{5.2}$$

Now we can define the waste rectangle WR with dimensions (h_{WR}, w_{WR}) for a compound C depending on the type of the compound:

$$(h_{WR}, w_{WR}) := \begin{cases} (h_C^{\max}, \tilde{w}_C), & \text{if } C \text{ is a horizontal compound.} \\ (\tilde{h}_C, w_C^{\max}), & \text{if } C \text{ is a vertical compound.} \end{cases}$$

5.4 Objective value

As in the problem definition of K2DCSPV stated, the main objective is to minimize the costs of a solution. But to be able to guide the search and to tie-break different solutions with the same costs we introduce additional secondary objectives. The idea is that if two solutions have the same cost and possibly use exactly the same sheet types we want to favor the one solution which has a pattern which is almost empty. We want to favor such a solution since we hope to be able to remove an almost empty pattern by improving our solution and therefore reducing the costs. Clearly we cannot just sum over the wastes since if both solutions contain all elements and use the same sheet types then the sum of wastes will be the same. By squaring the waste ratios of each pattern we favor solutions where the wastes are not distributed across multiple sheets but concentrate on few or even only one sheet. We therefore introduce as secondary objective the sum of squared waste ratios

$$c_2(S) := \frac{\sum\limits_{P \in \mathcal{P}^S} c_{t_P} w(P)^2}{\sum\limits_{P \in \mathcal{P}^S} c_{t_P} a_P^S}$$

where $w(P)$ denotes the waste ratio of the pattern P. For a pattern P the waste ratio is defined by

$$w(P) := \frac{H_t W_t - \sum_{i \in E} e_i^P h_i w_i}{H_t W_t} \tag{5.3}$$

where H_t and W_t are the dimensions of the used sheet type t and e_i^P is the element containment vector of the pattern P. Note that we squared waste ratios by the costs of the associated sheets and divide the whole sum of weighted squared waste ratios by the cost of the solution so that we get a value between 0 and 1. We want either to maximize the sum of squared waste ratios $c_2(s)$ or to minimize $1 - c_2(s)$, which is still between 0 and 1. We can use this fact to scale the objective such that it is always minor to the main objective, the costs of the solution. As scaling factor, we use the minimum cost of all sheet types. Therefore, we always prioritize a solution with one sheet less, regardless of the waste ratios.

All in all we get as total objective

$$\min o(S) := \sum_{P \in \mathcal{P}^S} a_P^S \cdot c_{t_P} + \left(1 - \frac{\sum\limits_{P \in \mathcal{P}^S} c_{t_P} w(P)^2}{\sum\limits_{P \in \mathcal{P}^S} c_{t_P} a_P^S}\right) \cdot \min_{t \in T} c_t.$$

5.5 Ruin Methods

In this section we will present different ruin methods used in the approach of Dusberger and Raidl [DR14]. They are then combined with different recreate methods, which we will present in the next section, and embedded in a variable neighborhood framework.

Each ruin method removes some nodes from the cutting tree. How to select the nodes to remove depends on the different ruin methods. All ruin methods have a parameter for the number of decrements δ or a percentage π which indirectly specifies δ by multiplying π with the total amount of possible nodes to remove, considering also the amounts of the nodes.

5.5.1 Ruin Random Subtree

This simple form has two variants, either it targets sheet patterns for removal or it targets leaf nodes for removal. We call the first variant ruin random sheet and the second ruin random element. In both variants the nodes for removal are selected uniformly random out of the pool of target nodes. Nodes can be selected multiple times but not more often than their amount value a. For each selection of a node its amount gets reduced by one and if it gets zero the node gets removed from the pattern tree.

5.5.2 Ruin by Maximum Waste Ratio

This ruin method is applied to sheet patterns. As defined in Section 5.4 we can associate with each sheet pattern P a waste ratio $w(P)$, see (5.3). The waste ratio basically tells us how dense the sheet type is filled with elements. The smaller the waste ratio the denser the sheet type is filled. Ideally we would like to remove patterns with high waste ratios since they are not filled densely, but we want to do that in a randomized way.

To do that we order all sheet patterns, i.e. level 1 nodes, by non-increasing waste ratio. To select one of the sheet patterns for removal we apply Algorithm 9. It uses a parameter

Algorithm 9 Randomized selection of max waste ratio patterns

INPUT: A partial solution S
OUTPUT: A sheet pattern of S which should get removed

1: **for** sheet pattern P in S in non-increasing waste ratio order **do**
2: With probability $1 - (1 - \text{ruin}_p)^{a_P}$ **return** P
3: **end for**
4: **return** pattern P with the highest waste ratio

ruin_p which determines the probability that the next pattern gets selected. After selecting a sheet pattern reduce its amount by one and remove it from the tree if the amount is zero. Repeat that until δ sheet patterns got removed.

5.5.3 Ruin and Merge

The idea of ruin and merge is to increase the amount of good patterns, i.e. patterns with low waste ratios. First a sheet pattern with a low waste ratio gets selected in a randomized way: We shuffle all sheet patterns, select the first one and iterate through the others in the shuffled order. Whenever we find a sheet pattern with a smaller waste ratio as the current, we set it to the current with a probability of 75%. The sheet pattern P_0 resulting from this procedure is then considered fixed.

In the next step we compute the total demand d_i^{tot} of all other sheet patterns, assume that S is the set of all sheet patterns of the current solution:

$$d_i^{\text{tot}} = \sum_{P \in \mathcal{P}^S \setminus \{P_0\}} e_i^P \cdot a_P^S \quad \forall i \in E.$$

Then the maximum value x for increasing the amount of P_0 gets calculated by

$$x := \min_{i \in E, e_i^{P_0} > 0} \left\lfloor \frac{d_i^{\text{tot}}}{e_i^{P_0}} \right\rfloor.$$

We increase a_{P_0} by x, i.e. $a_{P_0} \leftarrow a_{P_0} + x$ and remove enough other sheet patterns from S to do not overproduce any elements.

5.6 Construction Methods

In this section we will give an overview of the construction methods by Dusberger and Raidl. Since going too much into detail is out of scope for this work, an interested reader may read details in [DR14, DR15, DR17]. Construction methods are used for constructing a starting solution at the beginning and also for repairing partial solutions. We will present in this section different fill methods, each of them gets a partial solution S and tries to add elements to the patterns of the partial solution, i.e. tries to fill the partial solution. But they never add new sheets, they only fill the already existing patterns in S.

We then use such a fill method within a beam search framework where in each level of the search tree one new empty sheet gets added. After adding the sheet the fill method is applied for filling everything. We do that for each possible sheet type t, therefore the selection which sheet type to use next is not done greedily but in a sophisticated beam search approach. See Algorithm 5 for a generic overview how beam search works. Let us call the used fill method FILL, then the whole construction algorithm is given by Algorithm 10.

In the following we will present the different fill methods.

5.6.1 Fill Methods

All fill methods except for the dynamic programming approach have the same structure. In each iteration they insert a grid of elements of one type into a waste rectangle. The

Algorithm 10 Generic construction method using beam search

INPUT: A partial solution S_0
OUTPUT: A completed solution which extends S_0

```
S ← FILL(S)
Initialize the set of current partial solutions 𝒮 ← {S_0}
while 𝒮 is not empty do
    for S ∈ 𝒮 do
        for t ∈ T do
            if Σ_{P∈𝒫^S : t_P = t} a_P^S < q_t then
                Clone S to S' and add an empty pattern of sheet type t to S'
                S' ← FILL(S')
                Compute objective value o(S')
            end if
        end for
    end for
    Clear 𝒮
    Store the β best partial solutions S', according to o(S'), in 𝒮
    If any S' is complete store the best objective o^best and its solution
    Remove all partial solutions from 𝒮 which cannot be better than o^best
end while
return the best found complete solution
```

used element type must be in the set

$$E_R^S := \{i \in E : d_i^{\mathrm{r},S} > 0\}$$

where $d_i^{\mathrm{r},S}$ is the residual demand of the element type i in the partial solution S.

How they decide, which element type in E_R^S they use, which waste rectangle they use, and how big the grid is depends on the method. They stop when no element could get inserted anymore.

Critical Fit Insertion Heuristic

In each iteration the critical element type is calculated. The critical element type $i_{\mathrm{c}}^S \in E_R^S$ is defined by the properties

- $\nexists i \in E_R^S : i \neq i_{\mathrm{c}}^S \wedge w_i \geq w_{i_{\mathrm{c}}^S} \wedge h_i \geq h_{i_{\mathrm{c}}^S}$,
- the number of sheets in S which contain a waste rectangle into that the element i_{c}^S fits is minimal,
- if multiple element types satisfy the above conditions, use the one with the smallest index.

We now fix the critical element type $i := i_c^S$ and search for a waste rectangle WR of a compound C and a grid size $a_i^{\text{vert}} \times a_i^{\text{hor}}$ with $a_i^{\text{vert}} \cdot a_i^{\text{hor}} \leq d_i^{r,S}$, such that the insertion of the grid into the waste rectangle WR results in a maximal fitness. The fitness of an insertion with the given grid size into the compound C, resulting in a compound C', is defined by

$$f(C, a_i^{\text{vert}}, a_i^{\text{hor}}) := \frac{1}{(\tilde{h}_{C'} + 1) \cdot (\tilde{w}_{C'} + 1) \cdot (\eta(C, a_i^{\text{vert}}, a_i^{\text{hor}}) + 1)}$$

where $\tilde{h}_{C'}$ is the slack height and $\tilde{w}_{C'}$ the slack width of the new compound C' after inserting the grid. Furthermore, $\eta(C, a_i^{\text{vert}}, a_i^{\text{hor}})$ is defined by

$$\eta(C, a_i^{\text{vert}}, a_i^{\text{hor}}) := \begin{cases} \min_{P \in C} |h_P - a_i^{\text{vert}} \cdot h_i|, & \text{if } C \text{ is a hor. compound.} \\ \min_{P \in C} |w_P - a_i^{\text{hor}} \cdot w_i|, & \text{if } C \text{ is vert. compound.} \end{cases}$$

The value of η tells us how well the grid fits to the other subpatterns of C, since we want for a horizontal compound that all subpatterns have almost the same height and for a vertical compound that all subpatterns have almost the same width. When we found an optimal waste rectangle and grid size according to the fitness criterion we insert the grid into this waste rectangle and update all data structures (like $E_R^S, i_c^S, ...$).

Fill Heuristic Based on Average-Area Sufficiency

We define the average area $\bar{a}(X)$ of a multiset of elements X by the average area of all elements in X (counted with their occurrence in the multiset). Furthermore, a pattern, a grid of elements and E_R^S can be interpreted as multisets of elements. For E_R^S the element counts are according to the residual demands.

If we want to insert a grid G into a waste rectangle, which is part of a sheet pattern P, then we define the average-area sufficiency criterion for this insertion by

$$\bar{a}(G \cup P) \leq \bar{a}(E_R^S \cup P)$$

where the unions are given by interpreting everything as multisets of elements.

If we want to compare two grid insertions, we distinguish the following:

- If both insertions satisfy the criterion, then the one yielding less waste is better
- If only one satisfies the criterion, the one which satisfies it is better
- If both do not satisfy the criterion, the one which violates the criterion the least is better

Now in each iteration we use the waste rectangle, element type, and grid size which is the best according to the above comparison criterion.

Naive Fill Heuristic

This fast heuristic implements a simple first-fit approach. It chooses the element type in E_R^S which has the largest area, the first waste rectangle in post order where the chosen element fits into and computes the maximum grid size for this element type and waste rectangle.

Dynamic Programming

Dynamic programming is a well-known technique for computing a recursion, where certain subproblems occur multiple times in different branches of the recursion tree. The idea is to store the results of common subproblems and looking up the value in the storage table instead of recomputing it. We want to use the dynamic programming technique in our case to develop an exact algorithm, which fills a waste rectangle with elements such that the filled area is maximal. We apply then that dynamic program to the largest waste rectangle of each sheet pattern. Note that the most important case is, when a sheet pattern is empty and therefore only has one waste rectangle of the size of the sheet type.

To be able to formulate a recursive formula for our problem, we need to lift the demands restriction, i.e. that all elements can be placed on the sheet an unlimited number of times. If the result overproduces some elements, we can remove those elements from the pattern afterwards and can apply another fill method to fill the remaining space. Furthermore, we need to discretize the possible cutting positions, for further details see [DR15]. The discretization results in a finite set P of possible cutting points for heights and a finite set Q of possible cutting points for widths.

We use variables $V_k^v(x,y)$ and $V_k^h(x,y)$ for $(x,y) \in Q \times P$ and $k \in \{0,\dots,K\}$. The variables $V_k^v(x,y)(V_k^h(x,y))$ represents the area of the optimal cutting pattern with width at most x and height at most y which uses at most k cutting levels and starts with a horizontal(vertical) cut, i.e. the root compound of the cutting tree is a vertical(horizontal) compound.

The idea of the recursion is now that a pattern with a horizontal compound as root node and cutting level k can either have one child or multiple children. If it has one child, then this is a vertical compound with one cutting level $k-1$. Otherwise, we can split off the last child and get on the left side a horizontal compound with cutting level k and on the right side a vertical compound with cutting level $k-1$. Vertical root compounds can be handled analogously, and we get all in all for all $k > 0$ and $(x,y) \in Q \times P$ the recursive formulation

$$V_k^v(x,y) = \max\left(V_{k-1}^h(x,y), \max_{x' \in Q, x' < x}\left(V_k^v(x',y) + V_{k-1}^h(x-x',y)\right)\right),$$

$$V_k^h(x,y) = \max\left(V_{k-1}^v(x,y), \max_{y' \in P, y' < y}\left(V_k^h(x,y') + V_{k-1}^v(x,y-y')\right)\right).$$

Table 5.1: List of Ruin and Recreate Neighborhoods

Name	Ruin Method	π	Fill Method
$\mathcal{N}_1$	Maximum Waste Ratio	$\pi \in \mathcal{U}(0.05, 0.33)$	Naive Fill
$\mathcal{N}_2$	Maximum Waste Ratio	$\pi \in \mathcal{U}(0.05, 0.33)$	Critical Fit
$\mathcal{N}_3$	Random Sheet	$\pi \in \mathcal{U}(0.05, 0.33)$	Critical Fit
$\mathcal{N}_4$	Random Element	$\pi \in \mathcal{U}(0.05, 0.33)$	Critical Fit
$\mathcal{N}_5$	Random Sheet	$\pi \in \mathcal{U}(0.05, 0.33)$	Naive Fill
$\mathcal{N}_6$	Random Sheet	$\pi \in \mathcal{U}(0.05, 0.33)$	Dynamic Programming
$\mathcal{N}_7$	Ruin and Merge	–	Dynamic Programming

For $k = 0$ and $(x, y) \in Q \times P$ we get

$$V_0^v(x, y) = V_0^h(x, y) = \max\left(0, \max_{i \in \{0,\dots,n-1\}, h_i \leq y, w_i \leq x} (h_i \cdot w_i)\right).$$

With that recursion we can apply a dynamic program which is implemented as a bottom up approach. Clearly this approach is exponential in running time but for many small sheet patterns it is enough fast in practice and returns good sheet patterns. If the amount of discretization points is too large, we turn off all neighborhoods which use this fill method.

5.7 Variable Neighborhood Search for Solving K2DCSPV

In the previous two sections we described ruin methods and construction methods. In this section we give an overview how we combine them to get our ruin and recreate neighborhoods for our variable neighborhood search procedure. Additionally, we introduce some other neighborhoods, which are not based on ruin and recreate but also used for our algorithm. Finally, we give a total overview of the algorithm, including how to construct initial solutions.

Table 5.1 lists all ruin and recreate neighborhoods used in the approach by Dusberger and Raidl. As we can see for all neighborhoods which use the parameter π, the parameter is chosen uniformly randomly from the interval $[0.05, 0.33]$. Note that π is chosen for every ruin call independently and stays never fixed. Furthermore, for the construction methods of the neighborhoods we always use a beam width of $\beta = 1$, which means that we decide greedily which sheet type we use next.

Additionally to the seven ruin and recreate neighborhoods, the approach by Dusberger and Raidl uses three restructuring neighborhoods which do not change the objective of the solution but may improve the solution representation. We will only shortly list those three neighborhoods as they are not the main component of the algorithm.

- $\mathcal{N}_8$: Tries to simplify patterns by using three different rules, merging subcompounds, reduce compounds with only one child, and rotating a sheet.
- $\mathcal{N}_9$: Brings patterns into a normal form and recognizes sibling patterns of the same structure and unifies them by increasing the amount.
- $\mathcal{N}_{10}$: Merges equivalent sheets, which are sheets which use the same sheet type and contain exactly the same elements but with different pattern structures.

Now we have described the ten used neighborhoods in the variable neighborhood structure, what remains is how to construct the initial solution. For that we also use the construction methods, but this time we also use larger beam widths than 1, depending on the fill method. We run four different construction methods all starting from the empty partial solution and chose the best result as starting solution.

- Method $\mathcal{C}_1$ uses the critical fit insertion heuristic with beam width $\beta = 10$.
- Method $\mathcal{C}_2$ uses the fill heuristic based on average-area sufficiency with beam width $\beta = 10$.
- Method $\mathcal{C}_3$ also uses the fill heuristic based on average-area sufficiency with beam width $\beta = 10$, but this time we replace areas with a so called value-correction framework. This framework assigns each element a value instead of an area. Then it iteratively tries to improve the values such that the solution quality increases. At the beginning the values equal the areas of the elements.
- Method $\mathcal{C}_4$ uses the dynamic programming fill method with beam width $\beta = 1$.

Now we presented everything to describe the whole algorithm, which is done in Algorithm 11.

5.8 Considering Pattern Setup Costs

We are especially interested in a variant of the K2DCSPV which uses pattern setup costs. The algorithm by Dusberger and Raidl which we presented in this chapter is mainly designed for the K2DCSPV and does not consider pattern setup costs. Although we can always incorporate pattern setup costs in the solution objective, see Section 5.4, the solution quality is still not as good as without setup costs. This is also one of the reasons why we were interested in solving the CSSCP or CSSSCPE as a post processing in a first place, since the approaches we presented in Chapter 4 specifically consider pattern setup costs and therefore can improve the solution quality often as we will see in the computational results chapter.

In the following we state the whole problem formulation, which was already presented at the end of Section 2.2.

Algorithm 11 VNS for solving K2DCSPV by Dusberger and Raidl

INPUT: An instance of K2DCSPV
OUTPUT: A solution S

```
1: for i ∈ {1, 2, 3, 4} do
2:     Apply construction method C_i
3: end for
4: S ← best solution found by the four construction methods
5: i ← 1
6: while termination criterion is not satisfied do
7:     S' ← apply neigbhorhood N_i to S
8:     if S' has a better objective than S then
9:         S ← S'
10:        i ← 1
11:    else if i = ℓ then
12:        i ← 1
13:    else
14:        i ← i + 1
15:    end if
16: end while
17: return S
```

K-staged two-dimensional cutting stock problem with variable sheet size and pattern setup costs (K2DCSPVSC). Given a set of sheet types T with widths $W_t \in \mathbb{R}_+$, heights $H_t \in \mathbb{R}_+$, available quantities $q_t \in \mathbb{N} \cup \{\infty\}$, costs $c_t \in \mathbb{R}_+$, and stacking costs c_t^{S} for $t \in T$. Furthermore, let $s^{\max} \in \mathbb{N} \cup \{\infty\}$ be the maximum stacking size and E be a set of different element types. Each element type $i \in E$ has a width $w_i \in \mathbb{R}_+$, a height $h_i \in \mathbb{R}_+$, and a demand $d_i \in \mathbb{N} \setminus \{0\}$. A solution S to the problem is now a set of patterns $\mathcal{P}^S$ and for each pattern $P \in \mathcal{P}^S$ an amount a_P^S. Furthermore, each pattern $P \in \mathcal{P}^S$ is associated with a sheet type t_P. Each pattern describes how to cut output elements out of the associated sheet type only using guillotine cuts. We can associate with each pattern $P \in \mathcal{P}^S$ an element vector $(e_i^P)_{i \in E} \in \mathbb{N}^{|E|}$ which describes how often the i-th element occurs in the pattern P. A solution is feasible if all element demands are satisfied, i.e.

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot e_i^P = d_i \quad \forall i \in E,$$

and all available sheet quantities are not exceeded, i.e.

$$\sum_{P \in \mathcal{P}^S : t_P = t} a_P^S \leq q_t \quad \forall t \in T.$$

The problem is now to find a feasible solution which minimizes the costs

$$\sum_{P \in \mathcal{P}^S} a_P^S \cdot c_{t_P} + \left\lceil \frac{a_P^S}{s^{\max}} \right\rceil \cdot c_{t_P}^{\mathrm{S}}. \tag{5.4}$$

To solve this problem we can simply use the approach by Dusberger and Raidl and adapt the objective function accordingly. To improve the approach in this situation we propose to add a neighborhood $\mathcal{N}_1 1$ which uses the hybrid method presented in Section 4.5 for finding a neighbor. This special neighborhood does not try to improve the current solution, but tries to build a new solution using all patterns found so far as pattern set $\mathcal{P}$. Furthermore, as construction method, we can use any construction method presented in this chapter in Section 5.6.

CHAPTER 6

Computational Results

In this chapter we present computational results from applying our techniques to real world instances. We use 192 instances for K2DCSPVSC originating from real world applications for many use cases. The algorithm by Dusberger and Raidl described in Chapter 5 is used to generate instances for CSSCP and CSSSCPE. Afterwards we present a comparison of our approaches on those generated instances. Furthermore, we compare the performance of the algorithm by Dusberger and Raidl with the algorithm when we add our hybrid neighborhood.

The implementations are done in C++ and we use Gurobi 8.1 to solve our ILP approach. All tests were performed on a single core of an Intel Xeon E5-2640 v4 processor with 2.4GHz using at most 8GB RAM.

6.1 Instances

In this section we present the instances we will use for our tests. As instances for the original problem, K2DCSPVSC, we use 192 real world instances which got provided by LodeStar Technology. The instances describe heterogeneous situations arising in different applications. Most instances have only one sheet type, but there are instances with up to 145 different sheet types. Furthermore, the number of different element types ranges from 1 to 176. And if we sum up all element demands the demand sum ranges from 1 to 8511. The instances with demand sum equal to one are trivial to solve but are still part of our testing portfolio.

For each of the 192 real world instances we consider three variants. The first variant considers no setup costs at all and the variant is named "N". The second variant considers setup costs which have medium influence compared to the sheet costs, we call that variant "M". And the third variant considers setup costs of high influence compared to the sheet costs, we call that variant "H". Therefore, we get all in all $3 \cdot 192 = 576$ instances.

Table 6.1: List of Instance Groups

Group	Bounds for $\mathcal{P}$	#Instances N	#Instances M	#Instances H
G_1	$\lvert\mathcal{P}\rvert < 10$	71	74	72
G_2	$10 \leq \lvert\mathcal{P}\rvert < 100$	33	32	34
G_3	$100 \leq \lvert\mathcal{P}\rvert < 500$	15	13	15
G_4	$500 \leq \lvert\mathcal{P}\rvert < 2,000$	11	12	11
G_5	$2,000 \leq \lvert\mathcal{P}\rvert < 10,000$	11	12	12
G_6	$10,000 \leq \lvert\mathcal{P}\rvert < 50,000$	11	9	10
G_7	$50,000 \leq \lvert\mathcal{P}\rvert < 200,000$	12	18	17
G_8	$200,000 \leq \lvert\mathcal{P}\rvert < 300,000$	12	11	11
G_9	$300,000 \leq \lvert\mathcal{P}\rvert < 500,000$	9	9	8
G_{10}	$500,000 \leq \lvert\mathcal{P}\rvert$	7	2	2

6.1.1 Generating Instances for CSSCP and CSSSCPE

We used the algorithm by Dusberger and Raidl described in Chapter 5 to generate instances for CSSCP and CSSSCPE. For each of the 576 real world instances we generate one instance for CSSCP and CSSSCPE. We do this by applying the algorithm by Dusberger and Raidl on each instance with a time limit of one hour and collect all patterns occurring during the execution of the algorithm. The resulting pattern sets $\mathcal{P}$ range from 1 pattern to 1,384,811 patterns.

As expected the variants "M" and "H" generate in general fewer patterns than the variant "N", since pattern setup costs lead to patterns being used many times instead of many different patterns. We group the instances for each variant into ten groups based on the size of $\mathcal{P}$. Table 6.1 shows the size limits of $\mathcal{P}$ for each group and how many instances are part of the group for each variant.

Note that we generated these instances by running the algorithm once. Since this is a randomized algorithm, running it multiple times would potentially result in completely different sets $\mathcal{P}$.

6.2 Comparing Solution Approaches for CSSCP and CSSSCPE

In this section we will present a comparison of the performance of the approaches presented in Chapter 3 on the different instance groups as described in the previous section.

All test runs have a time limit of one hour after which the algorithm returns without a feasible solution. Note that all our algorithms are deterministic and therefore we only run every test only once. Since some algorithms are quite time-consuming and the number

of patterns $|\mathcal{P}|$ can be up to 1,000,000 we use a preprocessing which filters the patterns based on their greedy rating

$$r(P) := \max_{a \in \mathbb{N}} r^S(P, a)$$

where S is in this case the empty partial solution. In the following we will analyze the performances of the algorithms when filtering the pattern set by $|\mathcal{P}| \leq X$ for different values for X, which means that only the best X patterns remain in the pattern set $\mathcal{P}$ according to the rating $r(P)$.

6.2.1 Comparing ILP with the Greedy Appraoch and the Hybrid Approach

Table 6.2 shows the results when we use relatively small pattern sets $|\mathcal{P}| \leq X := 500$ for the problem CSSCP. It compares the ILP approach with the Greedy approach and the hybrid approach. Column "G" gives the group index of the instance set, column "V" the variant abbreviation and "#" the number of instances in this set. Furthermore, for each of the three algorithms ILP, Greedy, and the Hybrid, the columns "feas." represent the number of instances in the set for which the respective algorithm returned a feasible solution. Moreover, the columns "obj." represent the geometric mean of the relative objectives of the solutions, which is the objective of the solution computed by the set cover algorithm divided through the objective of the solution computed by the original algorithm of Dusberger and Raidl, when the set cover instance got created. Therefore, a value greater than 1.0 represents a worse solution and a value smaller than 1.0 represents a better solution. We use here the geometric mean, since the values are all relative and therefore a geometric mean represents the average percentage gain/loss compared to the original solutions. Last but not least the columns "t(s)" represent the median CPU running times over the instance set in seconds.

Note that the Hybrid does not solve the CSSCP formally as the other two algorithms do. Therefore, the direct comparison is not fair, since the hybrid is allowed to add any feasible pattern compared to the other two algorithms, which only add patterns from $\mathcal{P}$. Nevertheless we included the hybrid results in the table since it shows how much the greedy can be improved by hybridizing by a construction heuristic.

As we can see, the results of the greedy algorithm are worse than the ILP results, regardless of the instance size. The ILP can improve some instances compared to the original result, even in variant "N" without setup costs, although only few instances could be improved. For variant "M" and "H" with setup costs significantly more instances could be improved by the ILP approach. We also see that there are many instances for which the ILP and the greedy couldn't find a feasible solution. This is because of the filtering, which removes too many patterns, especially for large instances, possibly removing all patterns containing some element types.

The hybrid can compensate those feasibility problems by creating new patterns. It is able to solve all instances except for one in G_9 variant "H" feasibly. Furthermore, the

Table 6.2: Comparing ILP, Greedy and Hybrid with filter $|\mathcal{P}| \leq X := 500$ and no exact demands

			ILP			Greedy			Hybrid		
G	V	#	feas.	obj.	t(s)	feas.	obj.	t(s)	feas.	obj.	t(s)
1	N	73	**73**	**1.00000**	0.01	**73**	1.00758	0.01	**73**	1.00758	0.01
2	N	31	**31**	**1.00000**	0.01	**31**	1.02749	0.01	**31**	1.02674	0.01
3	N	14	**14**	**1.00000**	0.03	**14**	1.04896	0.01	**14**	1.04896	0.03
4	N	12	10	**0.99873**	0.05	10	1.11779	0.04	**12**	1.10151	0.07
5	N	11	9	**1.01905**	0.18	8	1.13303	0.14	**11**	1.07301	0.14
6	N	10	7	**0.99725**	0.87	7	1.14021	0.83	**10**	1.04867	0.87
7	N	14	3	**1.00162**	2.94	4	1.07954	2.97	**14**	1.02528	3.43
8	N	9	2	2.26640	9.72	2	2.67964	9.57	**9**	**1.03283**	9.55
9	N	9	1	1.18627	16.57	1	1.57516	13.93	**9**	**1.03896**	15.45
10	N	9	1	1.03333	36.48	1	1.40000	32.59	**9**	**1.03126**	36.05
1	M	71	**71**	**0.99721**	0.01	**71**	1.00436	0.01	**71**	1.00436	0.01
2	M	35	**35**	**0.95217**	0.01	**35**	0.99385	0.01	**35**	0.98869	0.01
3	M	12	**12**	**0.95397**	0.10	**12**	0.98068	0.01	**12**	0.97670	0.02
4	M	12	10	**0.97295**	0.36	10	1.05713	0.03	**12**	1.05187	0.04
5	M	12	10	**1.00364**	0.41	10	1.12505	0.14	**12**	1.04935	0.14
6	M	10	6	**1.00269**	0.78	6	1.23805	0.55	**10**	1.02632	0.55
7	M	18	8	1.01567	7.15	8	1.34714	3.76	**18**	**1.00416**	3.68
8	M	8	1	1.04444	13.24	1	1.38611	9.20	**8**	**0.98234**	9.13
9	M	12	2	1.66667	14.42	2	1.84257	14.62	**12**	**0.99906**	15.49
10	M	2	0	-	35.75	0	-	35.62	**2**	**1.06155**	36.64
1	H	72	**72**	**0.97769**	0.01	**72**	**0.97769**	0.01	**72**	**0.97769**	0.01
2	H	32	**32**	**0.83524**	0.02	**32**	0.85845	0.01	**32**	0.85596	0.01
3	H	17	**17**	**0.78715**	0.08	**17**	0.82515	0.01	**17**	0.85028	0.01
4	H	12	10	**0.89496**	0.42	10	0.95264	0.03	**12**	0.94015	0.04
5	H	9	5	0.91550	0.49	5	1.07336	0.13	**9**	**0.88655**	0.15
6	H	13	8	0.98434	0.69	8	1.22177	0.50	**13**	**0.87109**	0.56
7	H	15	3	**0.79822**	3.98	3	1.07307	3.70	**15**	0.92964	4.14
8	H	12	2	1.11209	15.22	2	1.42074	11.08	**12**	**0.91710**	11.17
9	H	8	0	-	15.49	0	-	15.92	**7**	**0.87274**	16.04
10	H	2	0	-	40.29	0	-	40.55	**2**	**0.93896**	41.82

hybrid outperforms the ILP for the larger instance sets, regardless of the setup costs. Note that the running times of the algorithms itself are neglectable for this case, since the bottleneck for the larger instances is the filtering. This is why all running times look similar, regardless which algorithm is used.

If we lift the filter condition, the ILP has problems to solve the larger instances to optimality within the given time limit. Although the running times of the greedy and the hybrid do not change that much. Table 6.3 shows the results for the ILP, greedy and hybrid without a filter.

Although the ILP reaches the time limit often for the large instances it still can solve many more instances feasibly compared to having a filter of $|\mathcal{P}| \leq X = 500$. Furthermore, we also see that the geometric means of the ILP solutions, and also of the greedy and hybrid solutions, are clearly better than in the other case.

The situation looks similar if we only allow exact demands. In this case all three algorithms solve different problems. The ILP solves the CSSCPE, the Greedy solves the CSSSCPE and the Hybrid solves the original problem K2DCSPVSC. This gives the Greedy approach more flexibility compared to the ILP approach. Nevertheless, the ILP outperforms the Greedy also in this situation. Table 6.4 shows the results of the ILP, Greedy, and Hybrid for the unfiltered case with exact demands. Note that for variant "N" without setup costs the results are similar to the results with no exact demands. This is because without setup costs every solution with no exact demands can easily be transformed into a solution with exact demands, by removing overproduced elements, without decreasing the objective. Therefore, having no exact demands is no advantage. For the variants "M" and "H" the solution qualities of the algorithms are worse than with no exact demands, which is because they are now more restricted in what solutions are allowed. Nevertheless, the ILP can find improvements in many cases and the hybrid outperforms again the ILP on the large instance sets.

6.2.2 Tuning the β parameter for the PILOT and Beam Search Approach

Before we compare the ILP, greedy, and hybrid with the PILOT and Beam Search approach, we want to analyze the influence of the parameter β on their performances and find the best parameter setting. As we saw already for the ILP, Greedy, and Hybrid, filtering the patterns in a preprocessing does decrease the solution quality significantly. Therefore, we focus for the pilot and the beam search approach on the unfiltered instances.

We start with the case of no exact demands. Figure 6.1 compares for each of the variants "N", "M", and "H" the geometric mean of the objectives, the number of feasible solutions, the number of improved solutions, and the running times for three different values $\beta = 10, 30, 100$. As we can see the geometric means of the objectives are closely together, regardless of the β value. The larger β value works well on medium-sized instances and the smaller β values on larger instances. For the small instances the algorithm often finds

Table 6.3: Comparing ILP, Greedy and Hybrid without filter and no exact demands

			ILP			Greedy			Hybrid		
G	V	#	feas.	obj.	t(s)	feas.	obj.	t(s)	feas.	obj.	t(s)
1	N	73	**73**	**1.00000**	0.01	**73**	1.00758	0.01	**73**	1.00758	0.01
2	N	31	**31**	**1.00000**	0.01	**31**	1.02749	0.01	**31**	1.02674	0.01
3	N	14	**14**	**1.00000**	0.02	**14**	1.04896	0.01	**14**	1.04896	0.02
4	N	12	**12**	**0.99894**	0.08	**12**	1.11118	0.04	**12**	1.11123	0.07
5	N	11	**11**	**0.99725**	0.37	**11**	1.07712	0.18	**11**	1.07712	0.16
6	N	10	**10**	**0.99731**	2.49	**10**	1.07699	1.13	**10**	1.06509	1.06
7	N	14	**14**	**0.98932**	16.56	**14**	1.03287	4.04	**14**	1.02647	4.49
8	N	9	**9**	**0.98139**	72.47	**9**	1.02547	11.94	**9**	1.02410	13.39
9	N	9	8	**1.00815**	500.04	8	1.07948	19.53	**9**	1.04895	19.85
10	N	9	**9**	1.01136	1116.13	**9**	1.03460	46.50	**9**	**1.00161**	47.17
1	M	71	**71**	**0.99721**	0.01	**71**	1.00436	0.01	**71**	1.00436	0.01
2	M	35	**35**	**0.95217**	0.01	**35**	0.99385	0.01	**35**	0.98869	0.01
3	M	12	**12**	**0.95397**	0.08	**12**	0.98068	0.01	**12**	0.97670	0.02
4	M	12	**12**	**0.96246**	0.30	**12**	1.02878	0.03	**12**	1.04210	0.04
5	M	12	**12**	**0.97260**	14.41	**12**	1.07114	0.19	**12**	1.04416	0.15
6	M	10	**10**	**0.97860**	1843.40	**10**	1.04364	0.66	**10**	1.01661	0.72
7	M	18	**18**	1.00138	3600.00	**18**	0.99284	5.00	**18**	**0.98429**	5.13
8	M	8	**8**	1.07495	3600.00	**8**	1.01359	12.68	**8**	**0.98278**	12.11
9	M	12	11	1.08796	3600.00	11	1.06636	20.28	**12**	**0.98814**	19.03
10	M	2	1	1.37500	1858.78	**2**	1.06155	50.00	**2**	**1.04628**	51.55
1	H	72	**72**	**0.97769**	0.01	**72**	**0.97769**	0.01	**72**	**0.97769**	0.01
2	H	32	**32**	**0.83524**	0.01	**32**	0.85845	0.01	**32**	0.85596	0.01
3	H	17	**17**	**0.78715**	0.08	**17**	0.82515	0.01	**17**	0.85028	0.02
4	H	12	**12**	**0.87374**	1.47	**12**	0.93285	0.03	**12**	0.93180	0.04
5	H	9	**9**	**0.83205**	669.91	**9**	0.86050	0.14	**9**	0.86050	0.14
6	H	13	**13**	**0.81510**	3600.00	**13**	0.86171	0.57	**13**	0.86573	0.68
7	H	15	**15**	**0.86497**	3600.00	**15**	0.90709	4.63	**15**	0.91073	4.97
8	H	12	11	0.97068	3600.00	**12**	0.96792	15.48	**12**	**0.90360**	15.99
9	H	8	**7**	1.07489	3600.00	**7**	0.91525	22.09	**7**	**0.83721**	21.69
10	H	2	1	1.13146	1862.63	**2**	0.95304	58.35	**2**	**0.89348**	60.39

Table 6.4: Comparing ILP, Greedy and Hybrid without filter and exact demands

			ILP			Greedy			Hybrid		
G	V	#	feas.	obj.	t(s)	feas.	obj.	t(s)	feas.	obj.	t(s)
1	N	73	**73**	**1.00000**	0.01	**73**	1.00758	0.01	**73**	1.00758	0.01
2	N	31	**31**	**1.00000**	0.01	**31**	1.02749	0.01	**31**	1.02674	0.01
3	N	14	**14**	**1.00000**	0.03	**14**	1.04896	0.01	**14**	1.04896	0.02
4	N	12	**12**	**0.99894**	0.09	**12**	1.11118	0.04	**12**	1.11123	0.07
5	N	11	**11**	**0.99725**	0.44	**11**	1.07712	0.17	**11**	1.07712	0.21
6	N	10	**10**	**0.99731**	3.38	**10**	1.07699	1.20	**10**	1.06509	1.12
7	N	14	**14**	**0.98932**	21.63	**14**	1.03287	3.74	**14**	1.02647	4.76
8	N	9	**9**	**0.99607**	951.46	**9**	1.02547	13.76	**9**	1.02410	14.20
9	N	9	7	**1.00014**	3600.00	8	1.07948	21.52	**9**	1.04895	21.41
10	N	9	**9**	1.13108	3600.00	**9**	1.03460	49.01	**9**	**1.00161**	50.31
1	M	71	**71**	**1.00000**	0.01	**71**	1.00886	0.01	**71**	1.00886	0.01
2	M	35	**35**	**0.99896**	0.01	**35**	1.03266	0.01	**35**	1.01816	0.01
3	M	12	**12**	**0.98508**	0.18	**12**	1.03428	0.01	**12**	1.03028	0.02
4	M	12	**12**	**0.99129**	0.64	**12**	1.03018	0.03	**12**	1.03018	0.04
5	M	12	**12**	**0.99697**	6.13	**12**	1.07454	0.16	**12**	1.02128	0.19
6	M	10	**10**	**0.99218**	51.51	**10**	1.04163	0.77	**10**	1.01410	0.80
7	M	18	17	1.08978	3600.00	**18**	1.01688	5.17	**18**	**0.99756**	5.63
8	M	8	6	1.21903	3600.00	**8**	1.03782	13.53	**8**	**0.98745**	12.98
9	M	12	11	1.45684	3600.00	11	1.08120	20.85	**12**	**0.99145**	19.87
10	M	2	**2**	1.84637	3600.00	**2**	1.10782	53.09	**2**	**1.03078**	47.12
1	H	72	**72**	**1.00000**	0.01	**72**	1.01246	0.01	**72**	1.01246	0.01
2	H	32	**32**	1.00000	0.01	**32**	1.00677	0.01	**32**	**0.97817**	0.01
3	H	17	**17**	**0.96568**	0.08	**17**	1.01698	0.01	**17**	1.01698	0.02
4	H	12	**12**	**0.97714**	9.50	**12**	1.01633	0.03	**12**	0.99729	0.04
5	H	9	**9**	**0.99767**	1197.17	**9**	1.03353	0.11	**9**	1.00561	0.15
6	H	13	**13**	1.04216	2505.24	**13**	0.97739	0.70	**13**	**0.96444**	0.68
7	H	15	**15**	1.19202	3600.00	**15**	1.00350	5.28	**15**	**0.98096**	5.51
8	H	12	10	1.71607	3600.00	**12**	1.12348	19.09	**12**	**0.97643**	17.66
9	H	8	7	1.98132	3600.00	7	1.06027	25.16	**8**	**0.95989**	23.89
10	H	2	1	1.74178	3600.00	**2**	1.10563	57.43	**2**	**0.96376**	57.13

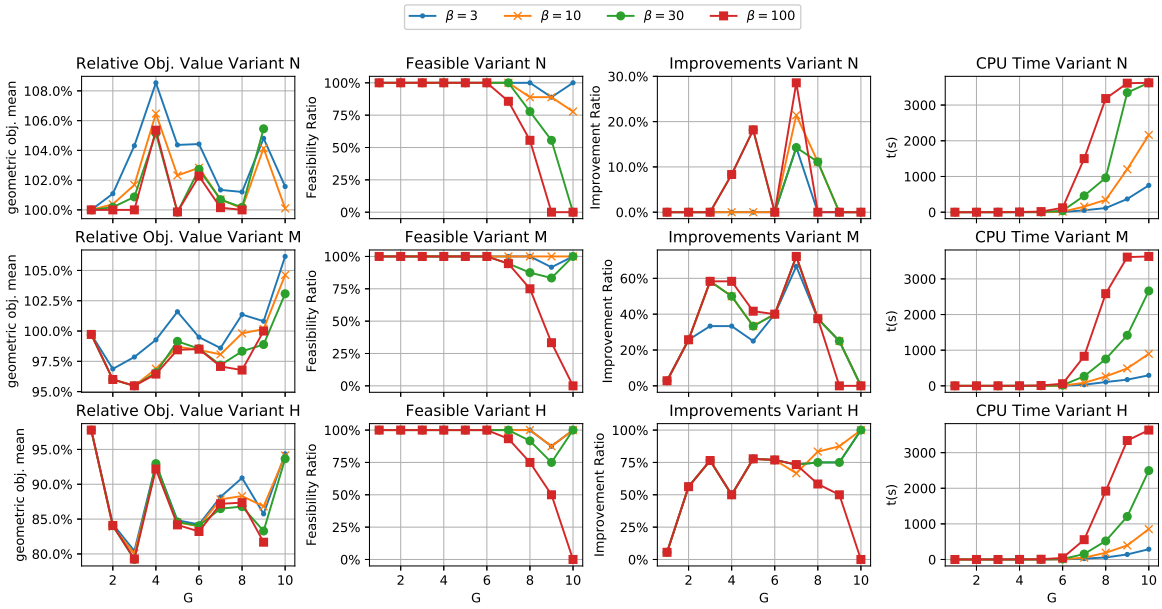

Figure 6.1: Comparison of three different β-values for the PILOT approach for unfiltered instances and no exact demands

a solution of the same quality regardless what β is. The same observation can be done in the case of exact demands, which is shown in Figure 6.2.

To verify observation we applied a Wilcoxon signed-rank test with p-value of 5%. All instance combinations using exact and not exact demands, variants "N", "M" and "H", but fix no filtering, are grouped together by the instance groups. Then on each instance group we compare the performance of each pair of the three *beta*-values. First of all the value $\beta = 3$ is significantly worse than $\beta = 10$ for the instance groups G_2 to G_9 and for the other two groups there is no significant difference. Because of that we won't consider $\beta = 3$ in the following and only compare $\beta = 10$ to $\beta = 30$ and $\beta = 100$. On the instance groups G_1, G_2, and G_3 none of the three β-values is significantly better than the other one. For group G_4 the values $\beta = 30$ and $\beta = 100$ are significantly better than $\beta = 10$. Furthermore, for group G_5 the value $\beta = 100$ is significantly better than $\beta = 10$ and for G_6 the value $\beta = 100$ is significantly better than $\beta = 30$. For group G_7, the value $\beta = 10$ is significantly better than $\beta = 100$, and for group G_8 both, $\beta = 10$ and $\beta = 30$ are significantly better than $\beta = 100$. Last but not least for the groups G_9 and G_{10} the value $\beta = 10$ is significantly better than $\beta = 30$ which in turn is significantly better than $\beta = 100$.

To always use a good parameter, we will fix the parameter for the PILOT by $\beta = 100$ for instances with $|\mathcal{P}| \leq 50,000$, i.e. all groups up to G_6, and $\beta = 10$ for instances with $|\mathcal{P}| > 50,000$.

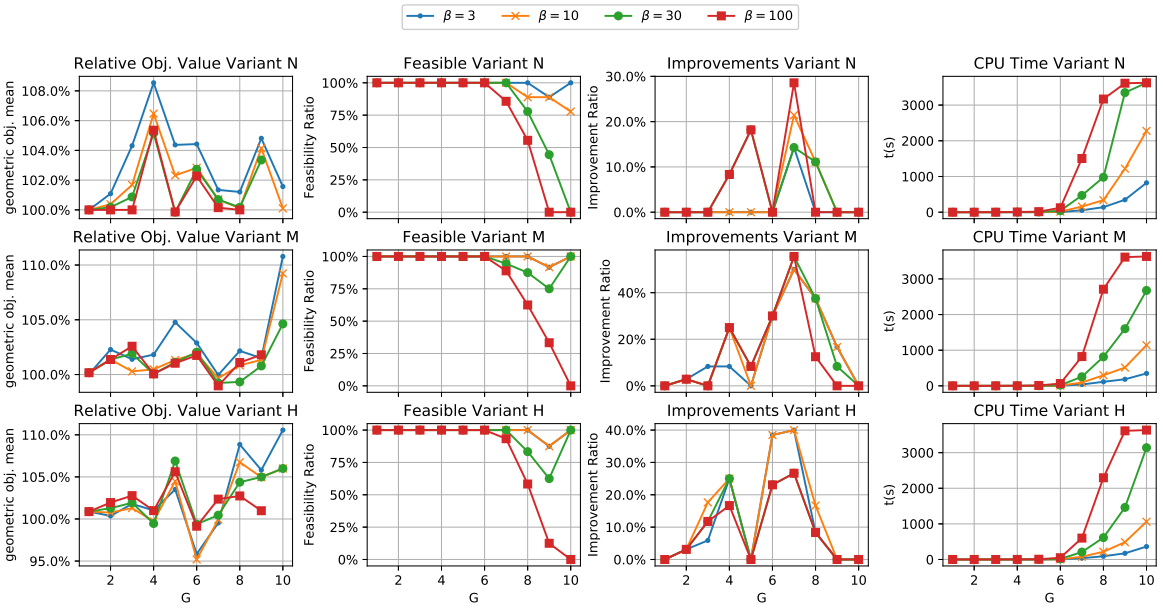

Figure 6.2: Comparison of three different β-values for the PILOT approach for unfiltered instances and no exact demands.

To tune the β-parameter for the beam search we proceed in the same way as for the PILOT approach. Figure 6.3 shows the results for the case of no exact demands and no filtering. As we can see the results are similar than for the PILOT approach. The large beam widths work well on medium-sized instances and the smaller beam widths well on the larger instances. Note that already for the instances of the instance group G_7 the beam search with beam width $\beta = 100$ runs into the time limit of one hour quite often. For exact demands the differences between the β-values are similar as we can see in Figure 6.4

We apply again a Wilcoxon signed-rank test with a p-value of 5% for verifying significant differences. Again the value $\beta = 3$ is significantly worse than $\beta = 10$ for instance sets G_4, G_5, and G_7 and is on no instance set significantly better, although, as we can see in the charts, it can solve more instances feasibly for the largest instance set G_{10}. We therefore won't consider $\beta = 3$ in the following analysis. For G_1, G_2, and G_3 there are no significant differences between the three β-values. For G_4 the values $\beta = 100$ is significantly better than $\beta = 30$ which in turn is significantly better than $\beta = 10$. Furthermore, for G_5 the values $\beta = 100$ and $\beta = 30$ are significantly better than $\beta = 10$ and for G_6 the value $\beta = 30$ is significantly better than $\beta = 10$. For the larger instance groups G_7, G_8, G_9, and G_{10} we have that $\beta = 10$ is significantly better than $\beta = 30$ which in turn is significantly better than $\beta = 100$.

We will use the same optimized parameter settings as for the PILOT also for the beam

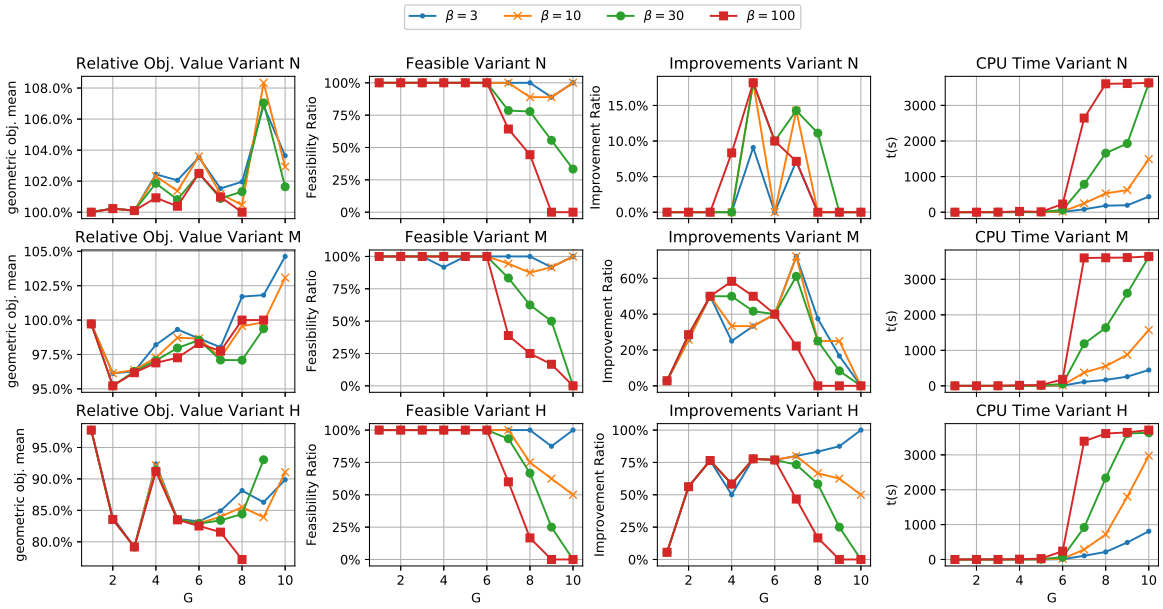

Figure 6.3: Comparison of three different β-values for the beam search approach for unfiltered instances and no exact demands

search approach. Therefore, if $|\mathcal{P}| \leq 50,000$ we use $\beta = 100$ and otherwise we use $\beta = 10$.

6.2.3 Comparing All Approaches

Finally, we are now able to compare all five approaches, the ILP, the greedy, the hybrid, the PILOT and the beam search approach. For the PILOT and beam search approach we use a β-parameter of $\beta = 100$ for instances with $|\mathcal{P}| \leq 50,000$ and $\beta = 10$ otherwise. Table 6.5 compares the results for all five approaches for no exact demands. The column "G" describes the group index of the considered instance group, the column "V" the variant, and the column "#" the number of instances for this group. Furthermore, for each of the five algorithms the column "f." lists the number of instances for which the algorithm found a feasible solution, "obj." gives the geometric mean of the objective values, only considering feasible solutions, the column "i." specifies the number of instances for which the algorithm found a solution which is better than the best solution found by the original algorithm when creating the instance, and the column "t(s)" the median running time in seconds.

We can see that the beam search and the PILOT approach can produce closely as good solutions as the ILP and for the larger instances even better solutions. Furthermore, they are often better than the hybrid or the greedy approach except for large instances. Note that especially for variant "H" the algorithms find for all instance groups except of the trivial group G_1 improvements in over half of the instances compared to the best

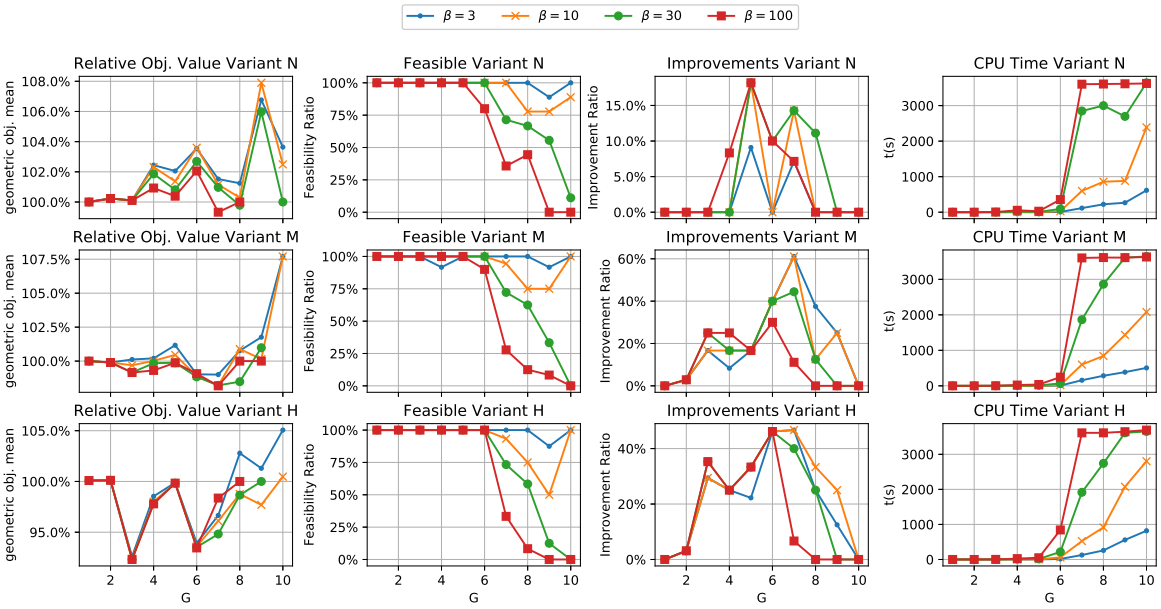

Figure 6.4: Comparison of three different β-values for the beam search approach for unfiltered instances and exact demands.

solution by the algorithm of Dusberger and Raidl. Therefore, our algorithm can really be used to improve the solution quality, especially but not only when setup costs are used.

Table 6.6 lists the results if we enforce exact demands. Note again that the ILP is more restricted than the other algorithms when we enforce exact demands. The results show similar properties, although not as many solutions could get improved as with no exact demands.

To compare the algorithms statistically we applied a Wilcoxon signed-rank test for each pair of algorithms grouping again all instances of each instance groups with both exact and not exact demands together. The ILP is significantly better than the greedy, PILOT, and the hybrid for the instance groups G_1 to G_6 and compared to the beam search it is significantly better on instance groups G_2 to G_5. On the other hand for instance groups G_8 to G_{10} the greedy and the hybrid are significantly better than ILP, on instance group G_7 to G_9 the pilot is significantly better than the ILP and finally the beam search is significantly better than the ILP on instance group G_7.

The greedy is significantly worse than the pilot on all instance groups except G_{10} and significantly worse than the beam search on instance groups G_1 to G_7. Compared to the hybrid it is significantly worse on the instance groups G_2 and G_5 to G_{10}. When we compare the PILOT approach with the beam search approach we get that the PILOT is significantly better for instance group G_9 and significantly worse on instance groups G_2 to G_7. Furthermore, the PILOT and the beam search approach are significantly better

Table 6.5: Comparing all five approaches without filter and no exact demands

			ILP				Greedy				PILOT				Beam Search				Hybrid			
G	V	#	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)
1	N	73	**73**	**1.000**	**0**	< 1	**73**	1.008	**0**	< 1	**73**	**1.000**	**0**	< 1	**73**	**1.000**	**0**	< 1	**73**	1.008	**0**	< 1
2	N	31	**31**	**1.000**	**0**	< 1	**31**	1.027	**0**	< 1	**31**	**1.000**	**0**	< 1	**31**	1.002	**0**	< 1	**31**	1.027	**0**	< 1
3	N	14	**14**	**1.000**	**0**	< 1	**14**	1.049	**0**	< 1	**14**	**1.000**	**0**	1	**14**	1.001	**0**	2	**14**	1.049	**0**	< 1
4	N	12	**12**	**0.999**	**1**	< 1	**12**	1.111	0	< 1	**12**	1.054	**1**	6	**12**	1.009	**1**	23	**12**	1.111	0	< 1
5	N	11	**11**	**0.997**	**3**	< 1	**11**	1.077	0	< 1	**11**	0.999	2	16	**11**	1.004	2	11	**11**	1.077	0	< 1
6	N	10	**10**	**0.997**	**4**	2	**10**	1.077	0	1	**10**	1.023	0	123	**10**	1.025	1	229	**10**	1.065	0	1
7	N	14	**14**	**0.989**	**8**	17	**14**	1.033	0	4	**14**	1.007	3	155	**14**	1.012	2	247	**14**	1.026	0	4
8	N	9	**9**	**0.981**	**5**	72	**9**	1.025	0	12	8	1.001	1	345	8	1.005	0	528	**9**	1.024	0	13
9	N	9	8	**1.008**	**4**	500	8	1.079	0	20	8	1.041	0	1204	8	1.083	0	620	**9**	1.049	0	20
10	N	9	**9**	1.011	**2**	1116	**9**	1.035	0	46	7	**1.001**	0	2164	**9**	1.029	0	1491	**9**	1.002	0	47
1	M	71	**71**	**0.997**	**2**	< 1	**71**	1.004	**2**	< 1	**71**	**0.997**	**2**	< 1	**71**	**0.997**	**2**	< 1	**71**	1.004	**2**	< 1
2	M	35	**35**	**0.952**	**10**	< 1	**35**	0.994	6	< 1	**35**	0.960	9	< 1	**35**	**0.952**	**10**	< 1	**35**	0.989	6	< 1
3	M	12	**12**	**0.954**	**7**	< 1	**12**	0.981	5	< 1	**12**	0.955	**7**	< 1	**12**	0.962	6	1	**12**	0.977	5	< 1
4	M	12	**12**	**0.962**	**8**	< 1	**12**	1.029	1	< 1	**12**	0.964	7	2	**12**	0.969	7	14	**12**	1.042	0	< 1
5	M	12	**12**	**0.973**	**6**	14	**12**	1.071	2	< 1	**12**	0.985	5	9	**12**	**0.973**	**6**	22	**12**	1.044	2	< 1
6	M	10	**10**	**0.979**	**4**	1843	**10**	1.044	**4**	1	**10**	0.985	**4**	59	**10**	0.983	**4**	181	**10**	1.017	**4**	1
7	M	18	**18**	1.001	7	3600	**18**	0.993	9	5	**18**	0.981	**13**	85	17	**0.973**	**13**	373	**18**	0.984	11	5
8	M	8	**8**	1.075	2	3600	**8**	1.014	3	13	**8**	0.998	3	264	7	0.996	2	553	**8**	**0.983**	**4**	12
9	M	12	11	1.088	1	3600	11	1.066	2	20	**12**	1.002	3	489	11	0.998	3	876	**12**	**0.988**	**6**	19
10	M	2	1	1.375	**0**	1859	**2**	1.062	**0**	50	**2**	1.046	**0**	897	**2**	**1.031**	**0**	1572	**2**	1.046	**0**	52
1	H	72	**72**	**0.978**	**4**	< 1	**72**	**0.978**	**4**	< 1	**72**	**0.978**	**4**	< 1	**72**	**0.978**	**4**	< 1	**72**	**0.978**	**4**	< 1
2	H	32	**32**	**0.835**	**18**	< 1	**32**	0.858	16	< 1	**32**	0.841	**18**	< 1	**32**	**0.835**	**18**	< 1	**32**	0.856	16	< 1
3	H	17	**17**	**0.787**	**13**	< 1	**17**	0.825	11	< 1	**17**	0.793	**13**	< 1	**17**	0.792	**13**	1	**17**	0.850	10	< 1
4	H	12	**12**	**0.874**	**9**	1	**12**	0.933	5	< 1	**12**	0.922	6	2	**12**	0.912	7	9	**12**	0.932	5	< 1
5	H	9	**9**	**0.832**	**7**	670	**9**	0.860	**7**	< 1	**9**	0.842	**7**	7	**9**	0.835	**7**	25	**9**	0.860	**7**	< 1
6	H	13	**13**	**0.815**	**11**	3600	**13**	0.862	9	1	**13**	0.832	10	44	**13**	0.825	10	239	**13**	0.866	9	1
7	H	15	**15**	0.865	**12**	3600	**15**	0.907	10	5	**15**	0.878	10	51	**15**	**0.840**	**12**	285	**15**	0.911	10	5
8	H	12	11	0.971	5	3600	**12**	0.968	5	15	**12**	0.883	**10**	188	9	**0.855**	8	718	**12**	0.904	9	16
9	H	8	**7**	1.075	1	3600	**7**	0.915	6	22	**7**	0.868	**7**	395	5	0.839	5	1799	**7**	**0.837**	**7**	22
10	H	2	1	1.131	0	1863	**2**	0.953	**2**	58	**2**	0.941	**2**	854	1	0.911	1	2974	**2**	**0.893**	**2**	60

Table 6.6: Comparing all five approaches without filter and exact demands

			ILP				Greedy				PILOT				Beam Search				Hybrid			
G	V	#	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)	f.	obj.	i.	t(s)
1	N	73	**73**	**1.000**	**0**	< 1	**73**	1.008	**0**	< 1	**73**	**1.000**	**0**	< 1	**73**	**1.000**	**0**	< 1	**73**	1.008	**0**	< 1
2	N	31	**31**	**1.000**	**0**	< 1	**31**	1.027	**0**	< 1	**31**	**1.000**	**0**	< 1	**31**	1.002	**0**	< 1	**31**	1.027	**0**	< 1
3	N	14	**14**	**1.000**	**0**	< 1	**14**	1.049	**0**	< 1	**14**	**1.000**	**0**	1	**14**	1.001	**0**	5	**14**	1.049	**0**	< 1
4	N	12	**12**	**0.999**	**1**	< 1	**12**	1.111	0	< 1	**12**	1.054	**1**	6	**12**	1.009	**1**	54	**12**	1.111	0	< 1
5	N	11	**11**	**0.997**	**3**	< 1	**11**	1.077	0	< 1	**11**	0.999	2	15	**11**	1.004	2	27	**11**	1.077	0	< 1
6	N	10	**10**	**0.997**	**4**	3	**10**	1.077	0	1	**10**	1.023	0	124	8	1.021	1	356	**10**	1.065	0	1
7	N	14	**14**	**0.989**	**8**	22	**14**	1.033	0	4	**14**	1.007	3	157	**14**	1.012	2	599	**14**	1.026	0	5
8	N	9	**9**	**0.996**	**3**	951	**9**	1.025	0	14	8	1.001	1	337	7	1.003	0	858	**9**	1.024	0	14
9	N	9	7	**1.000**	**3**	3600	8	1.079	0	22	8	1.041	0	1218	7	1.079	0	880	**9**	1.049	0	21
10	N	9	**9**	1.131	**1**	3600	**9**	1.035	0	49	7	**1.001**	0	2278	8	1.025	0	2387	**9**	1.002	0	50
1	M	71	**71**	**1.000**	**0**	< 1	**71**	1.009	**0**	< 1	**71**	1.002	**0**	< 1	**71**	**1.000**	**0**	< 1	**71**	1.009	**0**	< 1
2	M	35	**35**	**0.999**	**1**	< 1	**35**	1.033	0	< 1	**35**	1.014	**1**	< 1	**35**	**0.999**	**1**	< 1	**35**	1.018	0	< 1
3	M	12	**12**	**0.985**	**4**	< 1	**12**	1.034	0	< 1	**12**	1.026	0	< 1	**12**	0.992	3	2	**12**	1.030	0	< 1
4	M	12	**12**	**0.991**	**4**	1	**12**	1.030	0	< 1	**12**	1.001	3	2	**12**	0.993	3	22	**12**	1.030	0	< 1
5	M	12	**12**	**0.997**	**2**	6	**12**	1.075	0	< 1	**12**	1.010	1	10	**12**	0.999	**2**	38	**12**	1.021	**2**	< 1
6	M	10	**10**	0.992	3	52	**10**	1.042	3	1	**10**	1.017	3	64	9	**0.991**	3	245	**10**	1.014	**4**	1
7	M	18	17	1.090	4	3600	**18**	1.017	7	5	**18**	0.997	9	92	17	**0.982**	**11**	599	**18**	0.998	10	6
8	M	8	6	1.219	0	3600	**8**	1.038	3	14	**8**	1.008	3	296	6	1.009	1	842	**8**	**0.987**	**4**	13
9	M	12	11	1.457	0	3600	11	1.081	1	21	11	1.013	2	513	9	1.001	3	1436	**12**	**0.991**	**5**	20
10	M	2	**2**	1.846	**0**	3600	**2**	1.108	**0**	53	**2**	1.092	**0**	1142	**2**	1.077	**0**	2080	**2**	**1.031**	**0**	47
1	H	72	**72**	**1.000**	**0**	< 1	**72**	1.012	**0**	< 1	**72**	1.009	**0**	< 1	**72**	1.001	**0**	< 1	**72**	1.012	**0**	< 1
2	H	32	**32**	1.000	1	< 1	**32**	1.007	1	< 1	**32**	1.020	1	< 1	**32**	1.001	1	< 1	**32**	**0.978**	**3**	< 1
3	H	17	**17**	0.966	4	< 1	**17**	1.017	2	< 1	**17**	1.028	2	< 1	**17**	**0.923**	**6**	3	**17**	1.017	2	< 1
4	H	12	**12**	**0.977**	**3**	10	**12**	1.016	**3**	< 1	**12**	1.010	2	2	**12**	0.978	**3**	25	**12**	0.997	**3**	< 1
5	H	9	**9**	**0.998**	**3**	1197	**9**	1.034	0	< 1	**9**	1.056	0	7	**9**	**0.998**	**3**	50	**9**	1.006	0	< 1
6	H	13	**13**	1.042	3	2505	**13**	0.977	4	1	**13**	0.991	3	49	**13**	**0.935**	**6**	842	**13**	0.964	5	1
7	H	15	**15**	1.192	1	3600	**15**	1.003	5	5	**15**	0.999	6	77	14	**0.961**	7	525	**15**	0.981	**8**	6
8	H	12	10	1.716	0	3600	**12**	1.123	1	19	**12**	1.067	2	217	9	0.988	**4**	915	**12**	**0.976**	**4**	18
9	H	8	7	1.981	0	3600	7	1.060	0	25	7	1.050	0	486	4	0.977	2	2075	**8**	**0.960**	**5**	24
10	H	2	1	1.742	0	3600	**2**	1.106	0	57	**2**	1.060	0	1061	**2**	1.005	0	2804	**2**	**0.964**	**1**	57

Table 6.7: Instance properties of the instance set

Instance	Sheet Types	Element Types	Total Demand
1	1	30	153
2	1	10	59
3	1	20	132
4	1	15	56
5	1	7	36
6	1	64	278
7	3	21	8511
8	2	10	6660
9	3	60	86
10	1	176	522
11	1	50	2371
12	4	5	56
13	1	10	2000
14	1	41	130
15	1	97	1224
16	1	39	118
17	1	60	710
18	3	24	350
19	1	18	19
20	1	26	79

than the hybrid approach on instance groups G_1 to G_7 and significantly worse on the groups G_9 and G_{10}.

As we can see all algorithms have their advantages and disadvantages depending on the instance sizes compared to the others. For the smaller instances the ILP performs the best and the PILOT and beam search perform better than the greedy and hybrid. On the other hand for the large instances the fast greedy and hybrid approaches beat the other approaches.

6.3 Evaluating the Performance of the Hybrid Neighborhood

In this section we want to compare how adding the hybrid neighborhood as explained at the end of Section 5.8 changes the performance of the algorithm by Dusberger and Raidl. Since the set of 192 instances for the K2DCSPVSC is diverse we don't group them together but select 20 representative instances on which we compare the two algorithm variants. Table 6.7 shows the properties of those instances.

Since the algorithm by Dusberger and Raidl only considers exact demands, we also only allow exact demands, which has to be considered in the hybrid neighborhood. Since the base algorithm by Dusberger and Raidl is a randomized algorithm we run all our tests 30 times. Each run has a running time limit of one hour which is also always fully used, since time is the only termination criterion. Since the neighborhood should be fast, we use again filters to only keep at most the best X patterns during the execution. We tested the algorithm for different filters with $X = 500$, $X = 2000$, $X = 10000$, and $X = \infty$.

Table 6.8 shows the results for the instances 1 to 10 and table 6.8 for the instances 11 to 20. The column "I" is the instance number, "V" the variant and "#" the number of runs. Every further column represents the average objective over all runs for the different algorithm settings.

As we can see the original algorithm works quite well for the variant "N" without setup costs. Although for the other two variants, including the hybrid neighborhood often leads to better solutions. To verify that we apply a Wilcoxon signed-rank test on all pairs of algorithm settings for each variant. With that we could verify that the original algorithm without a hybrid neighborhood performs significantly better than all other algorithm variants for variant "N", i.e. with no setup costs. On the other hand for variant "H" all other algorithm variants perform significantly better than the original and for variant "M" there is no significant difference. Furthermore, the different X values have no significant differences.

Table 6.8: Performance of the neighborhood structures of the hybrid approach with different filters compared to original algorithm on instances 1 to 10

I	V	#	no hybrid	$\lvert\mathcal{P}\rvert \leq 500$	$\lvert\mathcal{P}\rvert \leq 2000$	$\lvert\mathcal{P}\rvert \leq 10000$	$\lvert\mathcal{P}\rvert < \infty$
1	N	30	**44.98**	**44.98**	**44.98**	**44.98**	**44.98**
2	N	30	**19.98**	19.99	19.99	19.99	19.99
3	N	30	**38.99**	**38.99**	**38.99**	**38.99**	**38.99**
4	N	30	**22.85**	22.86	22.86	22.86	22.86
5	N	30	**7.97**	**7.97**	**7.97**	**7.97**	**7.97**
6	N	30	**80.63**	81.93	81.93	81.93	81.93
7	N	30	**544.38**	547.85	548.05	548.16	547.91
8	N	30	**184.56**	185.18	185.17	185.06	185.20
9	N	30	**16.67**	17.38	17.38	17.40	17.40
10	N	30	**64.00**	65.26	65.42	65.39	65.19
1	M	30	29.50	**29.00**	**29.00**	**29.00**	**29.00**
2	M	30	**13.99**	**13.99**	**13.99**	**13.99**	**13.99**
3	M	30	25.73	25.41	25.33	25.29	**25.21**
4	M	30	**15.44**	16.08	16.13	16.10	16.18
5	M	30	**5.49**	**5.49**	**5.49**	**5.49**	**5.49**
6	M	30	**58.26**	60.45	60.27	60.87	60.95
7	M	30	286.65	283.09	283.50	**281.77**	282.95
8	M	30	98.96	98.93	99.03	98.83	**98.75**
9	M	30	15.17	**15.07**	15.14	15.14	15.14
10	M	30	48.49	48.36	48.41	48.44	**48.34**
1	H	30	14.60	**13.65**	**13.65**	**13.65**	**13.65**
2	H	30	8.54	**7.04**	7.10	**7.04**	**7.04**
3	H	30	**11.50**	**11.50**	**11.50**	**11.50**	**11.50**
4	H	30	**7.99**	8.15	8.15	8.15	8.03
5	H	30	**3.25**	**3.25**	**3.25**	**3.25**	**3.25**
6	H	30	28.91	27.74	27.66	27.97	**27.62**
7	H	30	40.55	**38.50**	38.65	38.67	38.63
8	H	30	16.14	**15.52**	15.57	**15.52**	15.53
9	H	30	**13.24**	13.26	13.26	13.26	13.25
10	H	30	29.01	29.03	28.96	28.95	**28.87**

Table 6.9: Performance of the neighborhood structures of the hybrid approach with different filters compared to original algorithm on instances 11 to 20

I	V	#	no hybrid	$\lvert\mathcal{P}\rvert \leq 500$	$\lvert\mathcal{P}\rvert \leq 2000$	$\lvert\mathcal{P}\rvert \leq 10000$	$\lvert\mathcal{P}\rvert < \infty$
11	N	30	**173.26**	180.64	181.34	180.88	180.35
12	N	30	**191.39**	193.99	193.93	193.99	193.96
13	N	30	**5.40**	5.41	5.41	5.42	5.41
14	N	30	**4.59**	4.73	4.73	4.72	4.72
15	N	30	**134.39**	137.56	137.66	137.69	137.56
16	N	30	**13.93**	13.95	13.95	13.95	13.95
17	N	30	**77.98**	81.03	80.97	81.00	80.90
18	N	30	**73.74**	74.03	74.05	74.04	74.04
19	N	30	**5.60**	5.74	5.80	5.80	5.74
20	N	30	**62.77**	**62.77**	**62.77**	**62.77**	**62.77**
11	M	30	101.73	101.54	101.36	101.66	**101.34**
12	M	30	100.28	100.01	100.13	100.06	**99.88**
13	M	30	4.76	4.76	4.76	**4.75**	4.76
14	M	30	**4.06**	4.36	4.36	4.36	4.34
15	M	30	83.06	82.36	82.33	**82.29**	82.61
16	M	30	10.98	**10.94**	10.96	10.96	10.96
17	M	30	48.57	48.17	48.49	48.25	**47.97**
18	M	30	43.72	43.48	43.50	**43.41**	43.56
19	M	30	5.07	5.11	5.11	5.11	**5.04**
20	M	30	**41.38**	41.40	41.43	**41.38**	41.40
11	H	30	23.86	22.03	21.83	**21.72**	22.25
12	H	30	17.08	**14.71**	14.90	14.93	14.77
13	H	30	**4.08**	**4.08**	**4.08**	**4.08**	**4.08**
14	H	30	**3.84**	4.04	3.97	4.04	4.04
15	H	30	27.62	26.71	26.63	26.52	**26.50**
16	H	30	7.55	7.43	**7.39**	7.42	7.43
17	H	30	15.39	**14.92**	15.09	14.94	15.14
18	H	30	14.55	13.01	**12.99**	13.01	13.07
19	H	30	4.74	**4.61**	4.91	4.88	4.71
20	H	30	22.20	22.20	**22.14**	**22.14**	22.26

CHAPTER 7

Conclusion

At the beginning of this thesis we explored the different types of cutting stock problems using the typology by Dyckhoff [Dyc90] and the extension of it by Wäscher et al [WHS07]. We presented some basic problems and also some variants of those, especially considering pattern setup costs. Despite the diversity of the different cutting stock problems we formulated a general problem GCSP which covers most of the previously presented problem variants.

Based on this GCSP we came up with a set covering problem CSSCP which receives as input a set of patterns and tries to find a solution for the GCSP which only uses a best subset of these patterns. In this way we omit the pattern construction part of the GCSP, which may be done with any problem specific solver for the underlying cutting stock problem, and only focus on the pattern selection part. We also always consider problem variants with exact demands compared to allowing overproduction.

To solve the set cover problem we propose five different approaches. The first one is an integer linear program which can solve the problem to optimality, although the running time increases exponentially for larger instances. Furthermore, we developed a greedy heuristic which rates different patterns based on the sum of area ratings of all elements on the pattern whose demand is not yet satisfied divided through the pattern costs. A sophisticated algorithm was developed for calculating the optimal amount for a pattern as fast as possible. We then used the greedy approach to extend it to a PILOT method which executes the greedy algorithm to rate an extension. Moreover, we used the greedy rating of patterns in a beam search which stores not only the best, but the best β partial solutions in each iteration. Last but not least we developed a hybrid method which uses the greedy approach in combination with a problem specific construction method which helps to improve the greedy if there are no suiting patterns left in the given instance pattern set. Note that the hybrid does not solve the CSSCP formally anymore, since it is able to construct new patterns which were not in the pattern set. Nevertheless, it is

still a valuable extension to the other algorithms, especially in practice since it is fast and often can find good solutions.

To test and apply our algorithms we presented the K-staged two-dimensional cutting stock problem with variable sheet size and pattern setup costs (K2DCSPVSC) which is a problem occurring in real world applications. We presented a sophisticated algorithm by Dusberger and Raidl [DR14, DR15, DR17] based on variable neighborhood search in combination with ruin-and-recreate based very large neighborhoods. They developed different ruin methods and many construction methods based on different greedy criteria but also one based on dynamic programming. We could then use this approach to generate pattern sets for different real world instances of the K2DCSPVSC which led to a set of testing instances for the CSSCP and its variant with exact demands.

Finally, we tested our algorithms on those generated instances. We applied all five algorithms to all instances and optimized the parameters, especially the β parameters for the PILOT and the beam search, by testing different values and applying a Wilcoxon signed-rank test for identifying significant differences. We then compared all five algorithms on different instances, grouped by their pattern set size. Again, we applied a Wilcoxon signed-rank test to identify significant differences. The algorithms perform differently depending on the instance sizes. For the small instances the ILP is significantly better than the others and the PILOT and beam search are significantly better than the greedy and the hybrid. On the other hand for the larger instances the greedy and the hybrid are significantly better than the other approaches. Especially if the pattern setup costs are high compared to the other costs the algorithms were able to improve over 50% of the interesting instances, i.e. the instances consisting of at least 10 patterns.

Last but not least we also tested the effects of incorporating the hybrid method as an own neighborhood search into the variable neighborhood search by Dusberger and Raidl. If there are no pattern setup costs including the new neighborhood leads to significantly worse solutions, since the neighborhood needs a lot of time and therefore reducing the amount of iterations which can be done within the time limit. On the other hand if the setup costs are high compared to the other costs the hybrid neighborhood leads to a significant overall improvement of the solutions.

For future work it would be interesting to apply our algorithms also to other cutting stock problems and to find out if it is also possible to improve algorithms for other problems. Furthermore, it would be interesting to develop improvement based or population based metaheuristics for the set covering problem and compare them with our approaches. Another idea would be to find good methods for producing a diverse set of patterns. In this thesis we used an algorithm by Dusberger and Raidl which was optimized for finding an as good solution as possible, but finding a diverse set of good patterns may lead to completely different algorithms.

List of Figures

List of Tables

List of Algorithms

List of Acronyms

1DCSP One-dimensional cutting stock problem, pp. 8, 9

1DCSPPR One-dimensional cutting stock problem with pattern reduction, p. 9

1DCSPSC One-dimensional cutting stock problem with setup costs, p. 9

1DMSSCSP One-dimensional multiple stock size cutting stock problem, p. 9

1DSSSCSP One-dimensional single stock size cutting stock problem, p. 9

2DGCSP Two-dimensional guillotine cutting stock problem, p. 10

2DICSP Two-dimensional irregular cutting stock problem, p. 10

2DOCSP Two-dimensional orthogonal cutting stock problem, p. 10

2DRCSP Two-dimensional regular cutting stock problem, pp. 9, 10

CSSCP Cutting stock set cover problem, pp. x, 15, 17–19, 24, 27, 29–31, 33–35, 37, 50, 53–55, 57, 59, 61, 63, 65, 71, 72

CSSCPE Cutting stock set cover problem with exact demands, pp. 15, 17, 18, 29, 57

CSSSCPE Cutting stock sub set cover problem for exact demands, pp. x, 29–31, 34, 37, 50, 53–55, 57, 59, 61, 63, 65

GCSP General cutting stock problem, pp. 13–15, 18, 29, 34, 35, 71

GCSPE General cutting stock problem with exact demands, pp. 14, 18, 29, 35

GVNS General variable neighborhood search, p. 39

ILP Integer linear program, pp. vii, 3, 17, 18, 55, 57, 62–66, 72, 75

K**2DCSP** K-staged two-dimensional cutting stock problem, pp. 11, 12

K**2DCSPV** K-staged two-dimensional cutting stock problem with variable sheet size, pp. ix, 12, 37, 40, 41, 43, 49, 50, 77

K2DCSPVSC K-staged two-dimensional cutting stock problem with variable sheet size and pattern setup costs, pp. 12, 50, 53, 57, 66, 72

PILOT Preferred iterative look ahead technique, pp. 30, 31, 77

RVNS Reduced variable neighborhood search, pp. 39, 40, 77

TSP Traveling salesperson problem, pp. 38, 40

VLNS Very large neighborhood search, p. 38

VND Variable neighborhood descend, pp. 38, 39

VNS Variable neighborhood search, pp. 38, 39, 50, 77

Bibliography

[AEOP02] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[AS80] A. Albano and G. Sapuppo. Optimal Allocation of Two-Dimensional Irregular Shapes Using Heuristic Search Methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(5):242–248, 1980.

[BPRR11] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

[BS07] G. Belov and G. Scheithauer. Setup and open-stacks minimization in one-dimensional stock cutting. *INFORMS Journal on Computing*, 19(1):27–35, 2007.

[CTF00] A. Caprara, P. Toth, and M. Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1-4):353–371, 2000.

[CZY15] Y. Cui, C. Zhong, and Y. Yao. Pattern-set generation algorithm for the one-dimensional cutting stock problem with setup cost. *European Journal of Operational Research*, 243(2):540–546, 2015.

[DC78] P. De Cani. A Note on the Two-Dimensional Rectangular Cutting-Stock Problem. *Journal of the Operational Research Society*, 29(7):703–706, 1978.

[DCVSN93] A. Diegel, M. Chetty, S. Van Schalkwyck, and S. Naidoo. Setup combining in the trim loss problem-3-to-2 & 2-to-1. *Durban: Business Administration*, 1993.

[DR14] 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 Maria J. Blesa, Christian Blum, and Stefan Voß, editors, *Hybrid Metaheuristics*, volume 8457 of *LNCS*, pages 85–99. Springer, 2014.

[DR15] 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.

[DR17] F. Dusberger and G. R. Raidl. A scalable approach for the k-staged two-dimensional cutting stock problem. In Karl Franz Dörner, Ivana Ljubić, Georg Pflug, and Gernot Tragler, editors, *Operations Research Proceedings 2015 — Selected Papers of the International Conference of the German, Austrian and Swiss Operations Research Societies (GOR, ÖGOR, SVOR/ASRO), University of Vienna, Austria, September 1–4, 2015*, pages 385–391. Springer, 2017.

[DV99] C. Duin and S. Voß. The Pilot method: A strategy for heuristic repetition with application to the Steiner problem in graphs. *Networks*, 34(3):181–191, 1999.

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

[FW00] H. Foerster and G. Wascher. Pattern reduction in one-dimensional cutting stock problems. *International Journal of Production Research*, 38(7):1657–1676, 2000.

[GG61] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

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

[HM99] P. Hansen and N. Mladenović. An Introduction to Variable Neighborhood Search. In Stefan Voß, Silvano Martello, Ibrahim H. Osman, and Catherine Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Springer US, Boston, MA, 1999.

[HM03] P. Hansen and N. Mladenović. A Tutorial on Variable Neighborhood Search. Technical Report G-2003-46, GERAD, 2003.

[HW13] S. Henn and G. Wäscher. Extensions of cutting problems: setups. *Pesquisa Operacional*, 33(2):133–162, 2013.

[KR18] Benedikt Klocker and Günther R Raidl. Solving a Weighted Set Covering Problem for Improving Algorithms for Cutting Stock Problems with Setup Costs by Solution Merging. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory – EUROCAST 2017*, Lecture Notes in Computer Science, pages 355–363, Gran Canaria, Spain, 2018. Springer International Publishing Switzerland.

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

[Low76] B. T. Lowerre. The HARPY Speech Recognition System. Technical report, Carnegie-Melon Univ Pittsburgh PA Dept of Computer Science, 1976.

[MCF+77] M. F. Medress, F. S. Cooper, J. W. Forgie, C. C. Green, D. H. Klatt, M. H. O'Malley, E. P. Neuburg, A. Newell, D. R. Reddy, B. Ritea, J. E. Shoup-Hummel, D. E. Walker, and W. A. Woods. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3):307–316, 1977.

[ME13] A. Mobasher and A. Ekici. Solution approaches for the cutting stock problem with setup cost. *Computers & Operations Research*, 40(1):225–235, 2013.

[PR10] D. Pisinger and S. Ropke. Large Neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, number 146 in International Series in Operations Research & Management Science, pages 399–419. Springer US, 2010.

[SP92] P. E. Sweeney and E. R. Paternoster. Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[WHS07] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European journal of operational research*, 183(3):1109–1130, 2007.

[YL05] J. Yang and J. Y.-T. Leung. A generalization of the weighted set covering problem. *Naval Research Logistics (NRL)*, 52(2):142–149, 2005.