Master's Thesis

Double Description Method in Cost Partitioning

Raphael Kübler*

August 29, 2022

University of Basel
M.Sc. Course in Mathematics

*Prof. Dr. Jérémy Blanc, Prof. Dr. Malte Helmert, Dr. Florian Pommerening

Acknowledgment

First of all, I would like to thank Prof. Dr. Jérémy Blanc and Prof. Dr. Malte Helmert, for making it possible for me to write this thesis as well as setting me up with Dr. Florian Pommerening.

Second, I would like to express my gratitude to Dr. Florian Pommerening. He supported and encouraged me throughout the four months of writing this thesis.

I also want to thank my family and friends for their support and for proofreading the final version of this thesis.

Last but not least I am grateful to my flatmate Raphael who turned out to be my own personal stackoverflow during the last month and helped me out when my computer broke a week before the deadline.

Abstract

Cost partitioning is a technique used to calculate heuristics in classical optimal planning. It involves solving a linear program. This linear program can be decomposed into a master and pricing problems. In this thesis we combine Fourier-Motzkin elimination and the double description method in different ways to precompute the generating rays of the pricing problems. We further empirically evaluate these approaches and propose a new method that replaces the Fourier-Motzkin elimination. Our new method improves the performance of our approaches with respect to runtime and peak memory usage.

Contents

Chapter 1

Introduction

Planning tasks are as old as humankind. They are concerned with finding a strategy to change the state of a given world into a desired state. We are able to change the state of the world by applying predefined actions. A strategy is a sequence of actions that we can apply. If this sequence changes the world into a desired state we call it *plan*. A naive way to find such a plan would be to search through all possible states our world could be in by considering the actions we can apply. Fortunately there exist more advances search methods that can limit the number of states we need to look at to find an optimal solution. In order to accomplish this, the search methods use functions named heuristics that compute a lower bound of for the distance of a state to the desired state. For example, they could tell us how many actions we need to apply at most to reach a desired state from our current state. In this thesis we are looking at a specific method to compute such heuristics. The method used is originally based in operations research and computing its value involves solving a large linear program with a technique called column generation. This technique relies on repeatedly computing solutions over a subset of constraints called the pricing problem. We attempt to solve this problem by precomputing all possible solutions to the price problem, although this seems to contradict the nature of column generation, as we will discover when we present the technique in detail. The reason we still explore this approach lies in the fact that there is reason to believe that the number of solutions is small in our practical cases (Pommerening et al., 2021). Moreover, this precomputation could potentially let us skip an expensive step in the computation of the heuristic functions. This would result in a faster search for a plan and would therefore let us solve the planning task more quickly.

In this thesis we compare two different approaches to precompute the solutions for the pricing problems. Moreover, we present a new method that replaces parts of the faster approach which further improves its running time.

To formally state our original problem, grasp the methods used to solve it and understand our two approaches knowledge of different concepts in computer science and math is

mandatory. The necessary background for computer science will be covered in chapter 2, whereas the mathematical background will be discussed in chapter 3. These concepts allow us to better understand the problem setting presented in chapter 4 and our theoretical results covered in chapter 5. In chapter 6 we discuss the results we got by implementing the different approaches and testing them on existing planning tasks from the International Planning Competitions[1].

The concepts presented throughout this thesis may sometimes seem challenging to understand. In order to convince you that they mostly just *seem* challenging, simple illustrating examples are provided along the way.

[1] http://ipc.icaps-conference.org

Chapter 2

Planning

Throughout most of this chapter, we will consider a logistics task as an example. This task takes place in the so called *logistics domain* introduced by McDermott (2000). An illustration of the example is given in Figure 2.1. In our domain there are two locations L_1 and L_2. At location L_1 there is a packet p and at location L_2 there are two trucks T_1 and T_2. The trucks are able to *drive* from one location to the other depicted by the arrow in Figure 2.1. Moreover, it is possible to *load* and *unload* the package from the trucks at both locations (given that the package is at the same location as the truck or that the package is inside the truck). The goal is to move the package from L_1 to L_2.

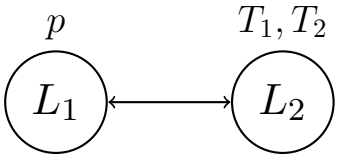

Figure 2.1: Illustration of the inital state of the logistics example.

2.1 Classical Optimal Planning

There are many different forms of planning. We will focus our attention on *classical optimal planning*. The word classical implies that our problem has the following three properties. It must be *static*, i.e. the state of our world only changes through our actions. Second, every action in our world only has one outcome. This property is called *deterministic*. Moreover, our world must be *finite* and *fully observable*, i.e. we only deal with a finite amount of states, actions and events and always have complete knowledge of the world. The word optimal refers to the fact that we only accept optimal plans as solutions. A plan is *optimal* if there does not exist another plan that is better with respect to the metric considered in

the problem. The metric we will consider is the smallest distance of a state to a desired state.

Formally describing a problem in classical optimal planning is not always done the same in research. We will use the *SAS*$^+$ *formalism* (Bäckström and Nebel, 1995). In order to define a planning task in this formalism, we need to introduce some concepts first.

In planning a *variable* v is an object that has a finite domain $dom(v)$. The set of all variables is denoted by $\mathcal{V}$. The process of assigning a value $w \in dom(v)$ to every variable $v \in \mathcal{V}$ is called *assignment*. We write $v \mapsto w$ to denote that the variable v is assigned the value w. A *partial assignment* γ maps a subset $vars(\gamma) \subseteq \mathcal{V}$ of the variables to $dom(vars(\gamma))$, where $vars(\gamma)$ is the set of all variables that are mapped to a value by γ. We refer to the value of a variable $v \in vars(\gamma)$ of a partial assignment γ by $\gamma[v]$.

A *state* s is a partial assignment over all variables, i.e. $vars(s) = \mathcal{V}$, and denote this by $s = \{v \mapsto w \mid w \in dom(vars(s))\}$. The set of all states is denoted by $\mathcal{S}$. We say that a state $s \in \mathcal{S}$ *satisfies* a partial assignment γ if $s[v] = \gamma[v]$ for all $v \in vars(\gamma)$. We then write $s \models \gamma$.

An action a is a tuple $\langle \text{pre}(a), \text{eff}(a), \text{cost}(a) \rangle$, where the *precondition* $\text{pre}(a)$ and the *effect* $\text{eff}(a)$ are partial assignments. By $\text{cost}(a_i)$ we denote the cost of applying action a. The set of all actions is denoted by $\mathcal{A}$. We say that an action $a \in \mathcal{A}$ is *applicable* in a state s if the precondition $\text{pre}(a)$ holds in s. Applying an action a in state s results in a new state $s' = s[\![a]\!]$ where $s[\![a]\!] \coloneqq \text{eff}(v)$ for all $v \in \text{eff}(a)$ and $s[\![a]\!] \coloneqq \text{eff}(v)$ for all $v \in \mathcal{V} \setminus \text{eff}(a)$.

We can now finally give a formal definition of a *SAS*$^+$ *planning task*.

Definition 2.1 (SAS$^+$ planning task) A SAS$^+$ planning task is defined as a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$, where

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of variables.
- $\mathcal{A} = \{a_1, \ldots a_m\}$ is a finite set of actions.
- $s_0 \in \mathcal{S}$ is the *initial state*.
- G is a partial assignment that defines the *goal states*. A state $s \in \mathcal{S}$ is called a goal state, if s satisfies G i.e. $s \models G$.

In the *SAS*$^+$ formalism states are given as a *factored representation*. This means that the states are encoded as assignments of the variables. We will take a look at our logistics example to get a better understanding on what the different concepts mean.

The set of variables is given by $\mathcal{V} = \{p, T_1, T_2\}$ and are mapped to their current location. Their respective domains are given in the following table along with the actions of our model for $1 \leq i, j, k \leq 2$.

variable	domain
p	$\{L_1, L_2, T_1, T_2\}$
T_1	$\{L_1, L_2\}$
T_2	$\{L_1, L_2\}$

action	precondition	effect
$drive(T_i, L_j, L_k)$	$T_i \mapsto L_j, L_j \neq L_k$	$T_i \mapsto L_k$
$load(p, T_i)$	$p \mapsto L_j, T_i \mapsto L_j$	$p \mapsto T_i$
$unload(p, T_i)$	$p \mapsto T_i, T_i \mapsto L_j$	$p \mapsto L_j$

The cost of every action is set to 1. The initial state is given by $s_0 = \{p \mapsto L_1, T_1 \mapsto L_2, T_2 \mapsto L_2\}$. Goal states must satisfy $G = \{p \mapsto L_2\}$.

2.2 State Space Search

A classical optimal planning task *induces a state space*[1]. The task of finding a plan becomes the task of finding a path from the initial state to a goal state.

Definition 2.2 (Induced State Space) The state space induced by a planning task Π is a 6-tuple $\Theta_\Pi = \langle \mathcal{S}, \mathcal{A}, \text{cost}, \mathcal{T}, s_0, G \rangle$, where

- $\mathcal{S}$ is the finite set of *states* of Π.
- $\mathcal{A}$ is the set of *actions* of Π.
- $\text{cost} : \mathcal{A} \to \mathbb{R}$ is the function that maps each action to a cost.
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of *transitions*. For $s, s' \in \mathcal{S}, a \in \mathcal{A}$ we have $\langle s, a, s' \rangle \in \mathcal{T}$ if and only if a is applicable in s and the effect of applying a in state s results in s'.
- $s_0 \in \mathcal{S}$ is the *initial state* of Π.
- $s_G \subseteq \mathcal{S}$ is the set of *goal states* of Π.

[1] Some authors use the term transition system instead of state space.

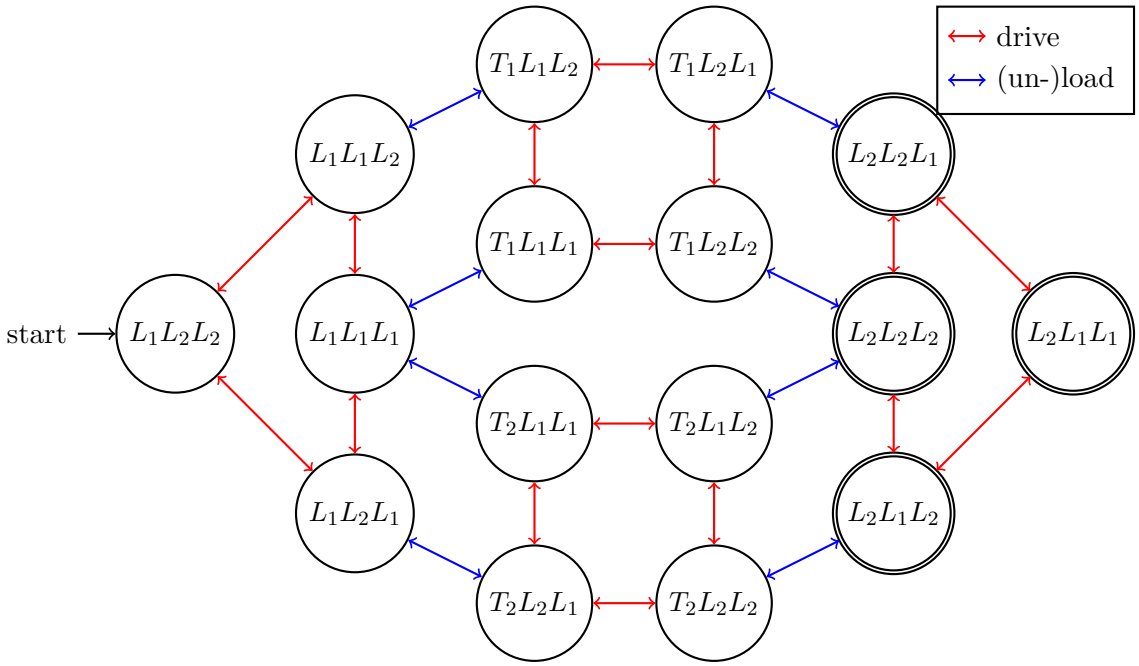

Figure 2.2: Illustration of the state space of the logistics example. We used the following abbreviations for the states $T_iL_jL_k = \{p \mapsto T_i, T_1 \mapsto L_j, T_2 \mapsto L_k\}$. The goal states are depicted by double-circles and the initial state is marked by start.

State space search can be grouped into two categories. The first category consists of *uninformed* search methods. Uninformed search methods systematically try to find a plan by only using the knowledge from the initial problem statement. *Informed* search methods use *heuristics* to approximate the distance from a state to the nearest goal state. Heuristic search is one of the state or the art search methods to solve classical optimal planning tasks.

Definition 2.3 (Heuristic) A heuristic h for a planning task Π with states $\mathcal{S}$ is a function $h : \mathcal{S} \times \text{cost} \to \mathbb{R} \cup \{-\infty\}$ that assigns a real value or $-\infty$ to every state $s \in \mathcal{S}$ considering the cost function cost. A heuristic h is called *perfect* and denoted by h^* if it maps every state $s \in \mathcal{S}$ to a cost of an optimal plan from s or to $-\infty$ if no plan exists. If cost is the cost function of the task we will simply write $h(s)$.

In our work we further restrict our heuristics. More precisely, we want our heuristic to be *admissible*. An admissible heuristic never overestimates the remaining cost of reaching a goal state, i.e. $h(s) \leq h^*(s)$, for all $s \in \mathcal{S}$. This property guarantees that if we use certain search-algorithms (e.g. A*-search) we will always find an optimal solution, if a solution exists.

A special family of heuristic search is *abstraction*. An abstraction α is a surjective func-

tion that maps states $s \in \mathcal{S}$ to *abstract states* $\mathcal{S}^\alpha$. If there was a transition between two states $s, s' \in \mathcal{S}$ there will also be a transition between the abstract states $\alpha(s), \alpha(s') \in \mathcal{S}^\alpha$. Formally, the abstraction function α *induces the abstract state space* Θ_{Π^α}.

Definition 2.4 (Induced Abstract State Space) Let Π be a planning task and Θ_Π its induced state space. Let $\alpha : \mathcal{S} \to \mathcal{S}^\alpha$ be an abstraction function. We call $\Theta_{\Pi^\alpha} = \langle \mathcal{S}^\alpha, \mathcal{A}, \text{cost}, \mathcal{T}^\alpha, s_0^\alpha, G^\alpha \rangle$ the induced abstract state space of α, where $\mathcal{S}^\alpha = \{\alpha(s) \mid s \in \mathcal{S}\}$, $\mathcal{T}^\alpha = \{\langle \alpha(s), a, \alpha(s') \rangle \mid \langle s, a, s' \rangle \in \mathcal{T}\}$, $s_0^\alpha = \alpha(s_0)$ and $G^\alpha = \{\alpha(s) \mid G \models s\}$.

Abstract state spaces are an important tool in practice because their state space is small enough to be stored in computer memory and every plan in the original state space is also a plan in the abstract state space. Additionally, the cost of an optimal plan in the original state space will always be higher or equal than the cost of an optimal plan in the abstract state space. Thus using the cost of the abstract is an admissible heuristic.

Theorem 2.1 Let Π be a planning task and $\Theta_\Pi = \langle \mathcal{S}, \mathcal{A}, \text{cost}, \mathcal{T}, s_0, s_* \rangle$ its corresponding state space. Let $\alpha : \mathcal{S} \to \mathcal{S}^\alpha$ be an abstraction function. A plan p in the original state space Θ_Π is also a plan in the abstract state space Θ_{Π^α}. Moreover, the optimal goal distances in the abstract state space are an admissible heuristic for the original state space.

A special case of abstractions are *projections*. For projections the abstraction function maps the states onto a *pattern*. A pattern is a subset of the original variables. Abstract state spaces can be computed quite fast if the patterns are small, since all preconditions and effects outside the pattern can be ignored.

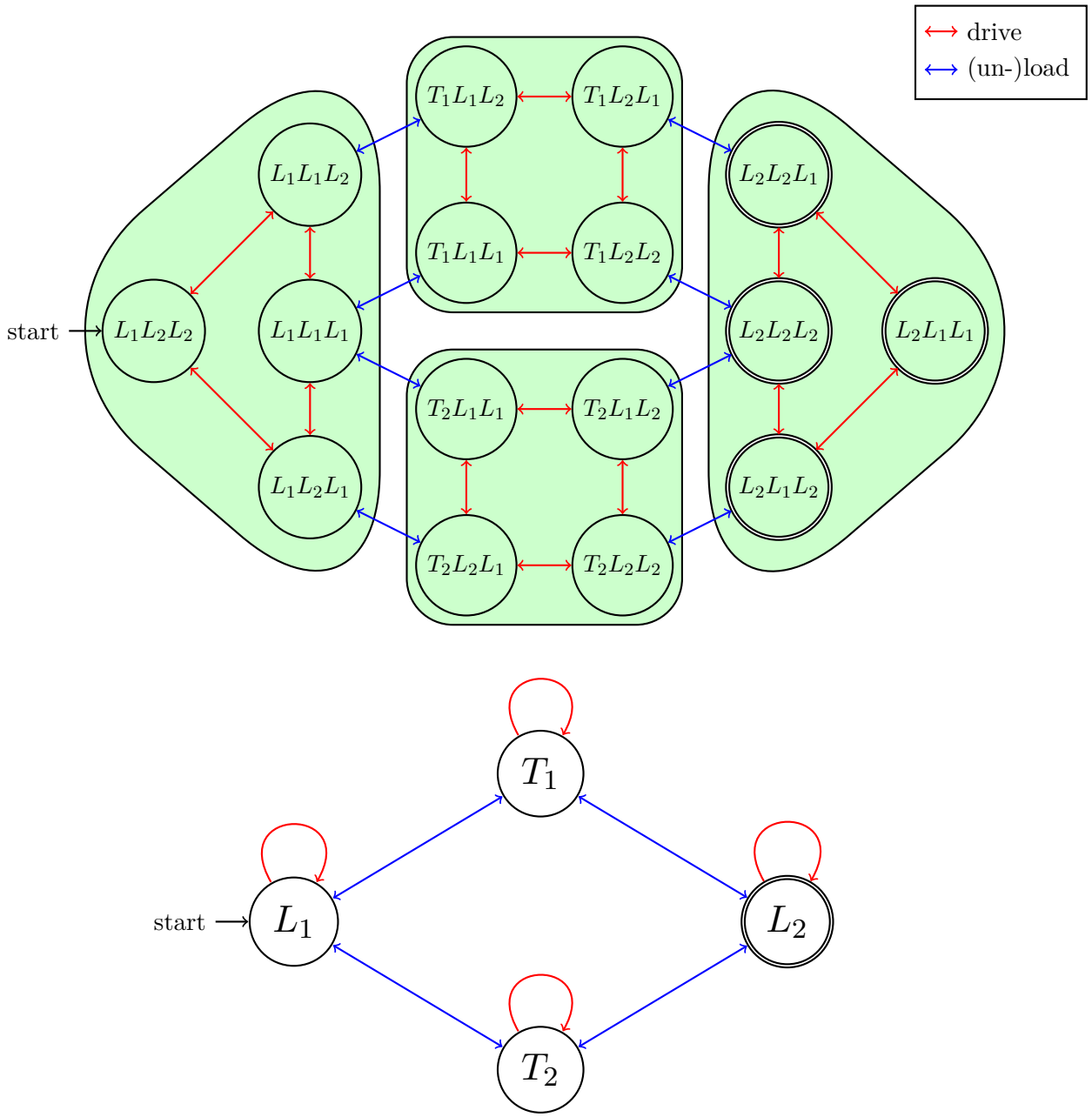

Figure 2.3: Illustration of the state space for the abstraction function that projects each state to the position of the packet. The top image shows the abstraction function in the original state space the bottom shows the induced state space.

2.3 Cost Partitioning

One of the problems of abstraction heuristics is that there are a lot of different admissible heuristics and it is usually not clear which one we should choose. Moreover, it was shown that using a single heuristic is usually not enough to capture all the details needed for solving the problem efficiently (Holte et al., 2006). It is therefore common to use multiple heuristic functions and combine them to generate a better estimate. A priori we do not know if the combination of multiple admissible heuristics is admissible as well. Furthermore, we do not know how to combine the results of each heuristic to obtain a better heuristic. This is where the idea of cost partitioning comes into play. Cost partitioning is a technique that distributes the original costs of the problem among the different heuristics, such that their combination is again admissible (Katz and Domshlak, 2010; Pommerening et al., 2015). In the following defintions and theorems we formally define *(optimal) cost partitioning* and show how to construct an admissible (optimal) cost partitioning.

Definition 2.5 (General Cost Partitioning; Pommerening et al., 2015) Let Π be a planning task with a set of actions $\mathcal{A}$ and cost function cost: $\mathcal{A} \to \mathbb{R}$. A general cost partitioning for Π is a tuple $P = \langle \text{cost}_1, \ldots, \text{cost}_n \rangle$ where $\text{cost}_i : \mathcal{A} \to \mathbb{R}$ for $1 \leq i \leq n$ and $\sum_{i=1}^{n} \text{cost}_i(a) \leq \text{cost}(a)$ for all $a \in \mathcal{A}$.

Theorem 2.2 (Pommerening et al., 2015) Let Π be a planning task and let $P = \langle \text{cost}_1, \ldots, \text{cost}_n \rangle$ be a general cost partitioning. For admissible heuristics $h_1, \ldots, h_n$ of Π we get that

$$h_P(h_1, \ldots, h_n, s) = \sum_{i=1}^{n} h_i(s, \text{cost}_i) \tag{2.1}$$

is an admissible heuristic for every state s of the planning task Π. Note that if any term in (2.1) is ∞, then the whole sum is defined as ∞, even if one of the terms is $-\infty$.

Definition 2.6 (Optimal General Cost Partitioning; Pommerening et al., 2015) Let Π be a planning task and let $\mathcal{P}_n$ be the set of all general cost partitionings for Π with n elements. We say that a general cost partitioning is optimal, if it achieves the highest heuristic value for a given state and heuristics. More formally, we define the set of all general optimal cost partitionings for admissible heuristics in a state s of Π as

$$\text{OCP}(h_1, \ldots, h_n, s) = \arg\max_{P \in \mathcal{P}_n} h_P(h_1, \ldots, h_n, s)$$

Theorem 2.3 (Pommerening et al., 2015) Let Π be a planning task and let $\mathcal{P}_n$ be the set of all general cost partitionings for Π with n elements. Moreover, let s be a state of Π. We can

calculate an optimal general cost partitioning estimate for admissible heuristics $h_1, \ldots, h_n$ in state s as follows

$$h^{\mathrm{OCP}}(h_1, \ldots, h_n, s) = \max_{P \in \mathcal{P}_n} (h_1, \ldots, h_n, s)$$

The method used to compute such cost partitions will be presented in chapter 4.

Chapter 3

Linear Inequalities

This chapter focuses on *systems of linear inequalities*. That is, given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ for some numbers $m, n \in \mathbb{N}$, we want to find vectors x such that $Ax \leq b$. Such vectors are called *feasible solutions*. The set of all feasible solutions of a system of linear inequalities is called *feasible set* or *polyhedron*. The polyhedron formed by a single linear inequality is called a *half-space*, since it divides the underlying space into two. Geometrically, we can therefore think of a polyhedron as the intersection of half-spaces. This is a very general object that can be grouped into different subclasses, as we will see later.

In this chapter we first discuss a technique for determining if solutions for general systems of linear inequalities exist. Afterwards we shift to the special case where the solution space is a *polyhedral cone*. We inspect some of its properties before giving a brief introduction into *linear programming*. We conclude the chapter by looking at two different approaches that help us solve large linear programs. Namely, we consider the *Dantzig-Wolfe Decomposition* that uses *column generation* for solving linear programs with a special structure.

3.1 Fourier-Motzkin Elimination

One of the most natural questions to ask for a system of linear inequalities is, if a feasible solution exists at all, i.e. if the feasible set is nonempty. Fourier and Motzkin independently developed a method to answer this question (Fourier, 1826; Motzkin, 1936). As we will see in just a moment, their method can not only be used to check if a solution to a system exists. It is particularly useful, if we want to reduce the number of variables in our system.

Suppose we are given a system of linear inequalities $Ax \leq b$ in n variables and we want to determine if it has a solution. The main idea of Fourier-Motzkin elimination is to reduce the question of feasibility of the original system in n variables, to one about a system in $n - 1$ variables. That is, the original system is feasible if and only if the reduced system is feasible. Fourier-Motzkin elimination achieves this by cleverly removing one of the n

variables such that the previous statement holds. Note that for a system of equalities, we could simply isolate one of the variables in any of the equations and then substitute it in all other equations.

We will look at a motivating example to understand how the method works and also get an intuition on *why* it works. Consider the following system of linear inequalities in 2 variables for which we would like to know if a feasible solution exists. An illustration is given in Figure 3.1 on the left side.

$$\begin{aligned} x_1 - 4x_2 &\leq -3 \\ -5x_1 + 2x_2 &\leq -3 \\ \frac{1}{2}x_1 + \ x_2 &\leq \frac{15}{2} \\ x_1 &\leq 7 \end{aligned}$$

For the sake of argument assume that the system is too large for us to be able to directly check if a feasible solution exists. Thus, we want to eliminate one of the variables and choose to eliminate x_2. Since we do not want to loose any information about the feasibility of the system, we try to see what we know about the feasible values of x_2. For this, we isolate x_2 in every inequality

$$\begin{aligned} x_2 &\geq \ \ \frac{1}{4}x_1 + \frac{3}{4} \\ x_2 &\leq \ \ \frac{5}{2}x_1 - \frac{3}{2} \\ x_2 &\leq -\frac{1}{2}x_1 + \frac{15}{2} \\ x_1 &\leq 7 \end{aligned} \tag{3.1}$$

We observe that there are three types of constraints with respect to x_2. The first constraint gives us a lower-bound restriction on a feasible solution for x_2. The next two constraints give us upper-bound restrictions and the last constraint gives us no restriction about feasible solutions for x_2. We will from now on group the constraints in *upper-bound*, *lower-bound* and *independent constraints*. We eliminate x_2 by combining every lower-bound constraint with every upper-bound constraint to create new constraints. The independent constraints should be preserved, since they give restrictions for the feasible solutions of the other variables.

Applying the described procedure to our example yields the following result.

$$\begin{aligned}\frac{1}{4}x_1 + \frac{3}{4} &\leq \frac{5}{2}x_1 - \frac{3}{2}\\ \frac{1}{4}x_1 + \frac{3}{4} &\leq -\frac{1}{2}x_1 + \frac{15}{2}\\ x_1 &\leq 7\end{aligned}$$

This is a system of linear inequalities in only one variable. Isolating the variables results in

$$\begin{aligned}x_1 &\geq 1\\ x_1 &\leq 9\\ x_1 &\leq 7\end{aligned}$$

For such a system it is quite easy to see, if a feasible solution exists. In our case this means that the system is feasible for $1 \leq x_1 \leq 7$, which we illustrate in Figure 3.1 on the right. We can plug these solutions into the previous system to calculate the feasible solutions for the variable that was eliminated in the step before. For example let us use $x_1 = 1$. The system (3.1) simplifies to

$$\begin{aligned}x_2 &\geq 1\\ x_2 &\leq 1\\ x_2 &\leq \frac{14}{2}\\ 1 &\leq 7\end{aligned}$$

and we have that $(1, 1)$ is a feasible solution for the original system and therefore the original system is feasible.

It is important to note that we could also have chosen to eliminate the last remaining variable by repeating the procedure of combining lower-bound and upper-bound constraints. This would have resulted in the two constraints $1 \leq 7$ and $1 \leq 9$. Since both constraints are satisfied this would have also implied that the original system is feasible.
To formalize this approach consider a general system of linear inequalities with m constraints and n variables.

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i, \quad i \in I \tag{3.2}$$

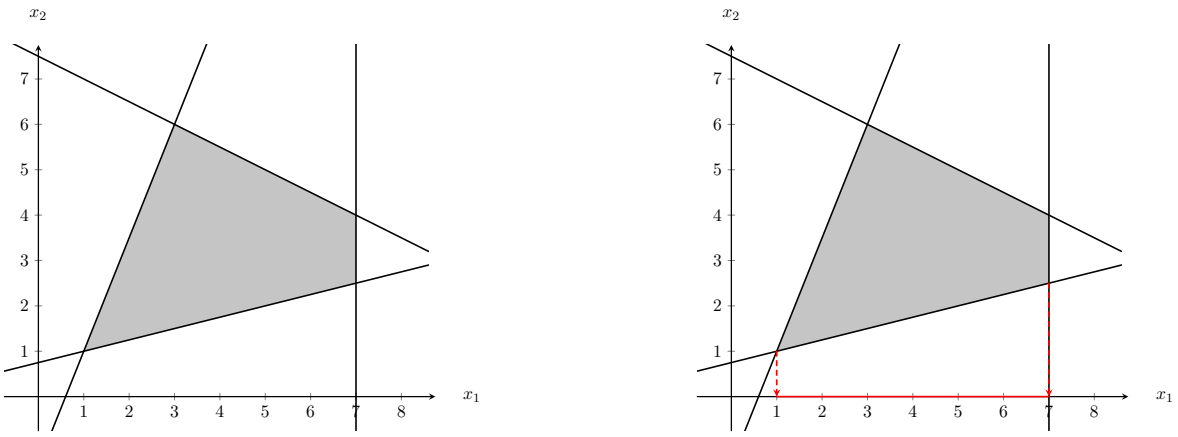

Figure 3.1: The left figure shows original system of linear inequalities. The feasible set is colored in grey grey. In the figure on the right we can see the result of Fourier-Motzkin elimination of variable x_2. It projects the feasible set onto the x_1-axis. The red arrows indicate the projection. The solid red line on the x_1-axis depicts the feasible set for x_1.

where $I = \{1, \dots, m\}$ denotes the index set for the constraints. We can split the index set into the following three groups

$$\begin{aligned} I^+ &= \{i \in I \mid a_{in} > 0\} \\ I^0 &= \{i \in I \mid a_{in} = 0\} \\ I^- &= \{i \in I \mid a_{in} < 0\} \end{aligned}$$

Consider now a second system that is a modification of (3.2).

$$\begin{aligned} \sum_{1 \le j \le n-1} (a'_{ij} + a'_{kj}) x_j \le b'_i + b'_k, \quad & i \in I^+, \ k \in I^- \\ \sum_{1 \le j \le n-1} a'_{ij} \ \cdot \ x_j \le b'_i, \quad & i \in I^0 \end{aligned} \tag{3.3}$$

where $a'_{ij} = \frac{a_{ij}}{|a_{in}|}$ and $b'_i = \frac{b_i}{|a_{in}|}$ for $i \in I^+ \cup I^-$ and $1 \le j \le n$.

Theorem 3.1 (Fourier-Motzkin elimination theorem; e.g. Conforti et al. (2014)) The following statement holds for general linear inequality systems as in (3.2)

$$\begin{aligned} & x' = (x_1, \dots, x_{n-1}) \text{ satisfies (3.3)} \\ \iff \ & \exists x_n \text{ s.t. } x = (x_1, \dots, x_{n-1}, x_n) \text{ satisfies (3.2)} \end{aligned}$$

3.2 Polyhedral Cones

The focus of this section is on the study of a special class of polyhedra that is considered throughout this thesis.

We say that a vector $x \in \mathbb{R}^n$ is a *linear combination* of vectors $x_1, \ldots, x_k \in \mathbb{R}^n$ if there exist scalars $\lambda_i, 1 \leq i \leq k$, such that

$$x = \sum_{i=1}^{k} \lambda_i x_i, \quad \lambda_i \in \mathbb{R}$$

A *conic combination* is a linear combination where we restrict the scalars λ_i to be non-negative. The *span* of a set of vectors $x_1, \ldots, x_k \in \mathbb{R}^n$ is given by the set of a linear combinations

$$\text{span}(x_1, \ldots, x_k) \coloneqq \left\{ \sum_{i=1}^{k} \lambda_i x_i \mid \lambda_i \in \mathbb{R} \right\}$$

Additionally we define the span of a matrix as the span of its column vectors. The *nullspace* of a matrix $A \in \mathbb{R}^{m \times n}$ is defined as the set of vectors $x \in \mathbb{R}^n$ such that $Ax = \mathbf{0}$, where $\mathbf{0}$ denotes the zero vector. We write $\text{Nul}(A)$ to denote the nullspace of a matrix A.

A set $C \subseteq \mathbb{R}^n$ is a *cone*, if it contains the origin and every positive multiple of any vector $x \in C$, i.e. $\lambda x \in C$ for all $\lambda \geq 0$. We say that C is a *convex cone* if every conic combination of some vectors is contained in C. The smallest cone containing a nonempty set $S \subseteq \mathbb{R}^n$ is denoted by $\text{cone}(S)$. By smallest we mean that there exists no cone C' such that $S \subseteq C' \subsetneq \text{cone}(S)$. The set S is said to *generate* $\text{cone}(S)$. For $S = \emptyset$, we define $\text{cone}(S) = \{\mathbf{0}\}$. For vectors $r_1, \ldots, r_k$, $k \geq 0$ we define $\text{cone}(r_1, \ldots, r_k) \coloneqq \text{cone}(S)$, where S denotes the set of all conic combinations of the vectors. Analogously we can define $\text{cone}(A)$ for a matrix A as the set of all conic combination of its column vectors.

A *finitely generated cone* is a convex cone C for which there exists a finite set of vectors $r_1, \ldots, r_k \in \mathbb{R}^n, k \geq 1$ such that $C = \text{cone}(r_1, \ldots, r_k)$. The vectors $r_1, \ldots, r_k$ are called the *generators* of C.

Definition 3.1 (Polyhedral Cone) We call a set $C \subseteq \mathbb{R}^n$ a *polyhedral cone* if it is the feasible set for a system of linear inequalities of the form $Ax \leq \mathbf{0}$ for a real matrix $A \in \mathbb{R}^{m \times n}$ and some positive numbers m, n, i.e. if

$$C = \{x \in \mathbb{R}^n \mid Ax \leq \mathbf{0}\}, \quad A \in \mathbb{R}^{m \times n} \tag{3.4}$$

This implies that a polyhedral cone is a nonempty polyhedron, since it always contains

the origin **0**. Moreover, it contains all *conic combinations* of vectors $x_1, \ldots, x_k \in C$. This implies that all polyhedral cones are convex, since every convex combination is also a conic combination. An example of a polyhedral cone is shown in Figure 3.2.

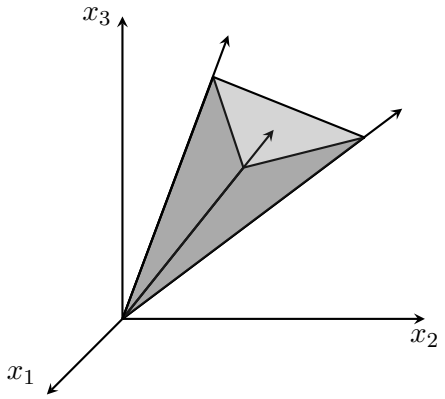

Figure 3.2: A polyhedral cone in 3-dimensions.

Now that we have defined our object of interest, we want to know more about it. We have described polyhedral cones as the intersection of a finite number of half-spaces containing the origin (3.4). The theorems of Minkowski and Weyl show, that there exists an alternative description for polyhedral cones.[1]

Theorem 3.2 (Minkowsky-Weyl Theorem for Polyhedral Cones; e.g. Conforti et al. (2014)) Let $S \subseteq \mathbb{R}^n$ be a set of points, then S is a polyhedral cone if and only if S is finitely generated.

A *ray* is a half-line that starts at a point $x_0 \in \mathbb{R}^n$ and moves in a direction d for infinity. More formally, any set $\{x \in \mathbb{R}^n \mid x = x_0 + \lambda d, \lambda \geq 0\}$ is called a ray. Hence the column vectors of R represent rays of cone(R), since $\{x \in \mathbb{R}^n \mid x = \lambda r, \lambda \geq 0, r \text{ is column vector of } R\}$ are rays of the cone. This may not seem like a big deal at first, but what it tells us is, that every polyhedral cone can be generated by a finite set of rays. This gives rise to many more questions such as: Given a cone how can we find rays that generate it? Are the generating rays unique?

Before answering these questions in the next section, we will turn our attention to a last special class of polyhedral cones, so called *pointed polyhedral cones*. The name comes from the fact that these polyhedral cones contain a point through which no ray passes, namely the origin. That is, if the cone contains a ray r, the negation $-r$ can not lie in the cone.

[1] There exists a more general formulation of this theorem for general polyhedra (Minkowski, 1910; Weyl, 1949).

For example the cone in Figure 3.2 is a pointed polyhedral cone. The rays r that have the property that their negation $-r$ is also contained in the cone form what is known as the *lineality space*.

Definition 3.2 (Lineality Space) The lineality space of a polyhedral cone $C = \{x \in \mathbb{R}^n \mid Ax \leq \mathbf{0}\}$ is given by

$$\text{lin}(C) = \{x \in \mathbb{R}^n \mid Ax = 0\} \tag{3.5}$$

We notice that the lineality space in the definition is equal to the nullspace of the matrix A. We can use the definition of the lineality space to define *pointed polyhedral cones*.

Definition 3.3 (Pointed Polyhedral Cone)[2] We call a polyhedral cone C pointed, if its lineality space only contains the origin, that is $\text{lin}(C) = \{0\}$.

Lemma 3.1 Let A be a matrix. We have that

$$\text{span}(A^T) \cap \text{Nul}(A) = \{\mathbf{0}\}$$

Proof. Let v be a vector in $\text{span}(A^T) \cap \text{Nul}(A)$. Since $v \in \text{span}(A^T)$ we have that there exists a vector x such that $v = A^T x$. Moreover, since $v \in \text{Nul}(A)$ we also have $Av = \mathbf{0}$. This lets make the following calculation

$$\begin{aligned} v^T v &= (A^T x)^T (A^T x) \\ &= x^T A A^T x \\ &= x^T A v \\ &= \mathbf{0} \end{aligned}$$

This implies that $v = \mathbf{0}$. □

We can use our two newly defined objects to generate any polyhedral cone by adding them together. The addition of two sets of vectors is given by their Minkowski sum.

Definition 3.4 (Minkowski Sum) Let U, V be two subset of $\mathbb{R}^n$. The *Minkowski sum* of U and V is given by

$$U + V \coloneqq \{u + V \mid u \in U, v \in V\}$$

Theorem 3.3 (Decomposition Theorem for Polyhedral Cones; e.g. Conforti et al. (2014))[3]

[2] The definition of the lineality space and the property of being pointed is the same for general polyhedra.
[3] This theorem is usually stated for general polyhedra (Conforti et al., 2014). However, this special case

Let $C = \{x \in \mathbb{R}^n \mid Ax \leq \mathbf{0}\}$ be a nonempty polyhedral cone. Then

$$C = \text{lin}(C) + Q \tag{3.6}$$

where Q is a pointed polyhedral cone and the addition denotes the Minkowski sum.

Proof. The reason this proof is included in the thesis is that it shows how to define the pointed polyhedral cone Q. An illustration of such a decomposition is given in Figure 3.3.

If C is pointed we have $\text{lin}(C) = \{\mathbf{0}\}$ and the statement follows for $Q \coloneqq C$. Thus we assume that C is not pointed. We first explicitly construct a pointed polyhedron Q and then show that $C = \text{lin}(C) + Q$ is true for our construction of Q. Let d denote the dimension of the lineality space of C. Since C is not pointed, d is greater than 0. Now let $B = \{b_1, \ldots, b_d\}$ be a basis of $\text{lin}(C)$, i.e. $\text{span}(B) = \text{lin}(C)$. We define $Q \coloneqq \{x \in \mathbb{R}^n \mid Ax \leq \mathbf{0}, B^T x = \mathbf{0}\}$. Thus we can rewrite Q as $Q = \{x \in \mathbb{R}^n \mid Dx \leq \mathbf{0}\}$ where

$$D = \begin{pmatrix} A \\ B^T \\ -B^T \end{pmatrix}$$

which is the definition of a polyhedral cone. Moreover, Q is also pointed, since we have

$$\begin{aligned}
\text{lin}(Q) &= \{x \in \mathbb{R}^n \mid Dx = \mathbf{0}\} \\
&= \{x_1 \in \mathbb{R}^n \mid Ax = \mathbf{0} \text{ and } B^T x = \mathbf{0}\} \\
&= \{x \in \mathbb{R}^n \mid Ax = \mathbf{0}\} \cap \{x \in \mathbb{R}^n \mid B^T x = \mathbf{0}\} \\
&= \text{lin}(C) \cap \text{Nul}(B^T) \\
&= \text{span}(B) \cap \text{Nul}(B^T) \\
&= \{\mathbf{0}\}
\end{aligned}$$

Where we used Lemma 3.1 in the last step. We are left to show that (3.6) holds for our defined Q. Since $\text{lin}(C) \subseteq C$ and $Q \subseteq C$ it follows that $\text{lin}(C) + Q \subseteq C$. To prove that $C \subseteq \text{lin}(C) + Q$, let $x \in C$. If $Ax = \mathbf{0}$, then $x \in \text{lin}(C)$ and we are done. If $Dx \leq \mathbf{0}$, then

suffices for our thesis.

$x \in Q$ and we are also done. Thus we may assume

$$Dx = \begin{pmatrix} b \\ c \\ -c \end{pmatrix}$$

with $Ax = b$ and $B^T x = c$. Since $x \in C$ and $x \notin \text{lin}(C)$ we have that $b < \mathbf{0}$. Since $Dx \leq \mathbf{0}$ we cannot have $c = \mathbf{0}$. By the orthogonality of x to $\text{lin}(C)$ and $b < \mathbf{0}$ there exists $y \in Q$ such that $Ay = b$ and $B^T y = \mathbf{0}$, i.e.

$$Dy = \begin{pmatrix} b \\ \mathbf{0} \\ -\mathbf{0} \end{pmatrix}$$

Moreover, let z be the orthogonal projection of x onto $\text{lin}(C)$, i.e. $z \in \text{lin}(C)$. We get

$$Dz = \begin{pmatrix} \mathbf{0} \\ c \\ -c \end{pmatrix}$$

Hence we have $Dx = Dy + Dz$ with $y \in Q$ and $z \in \text{lin}(C)$. □

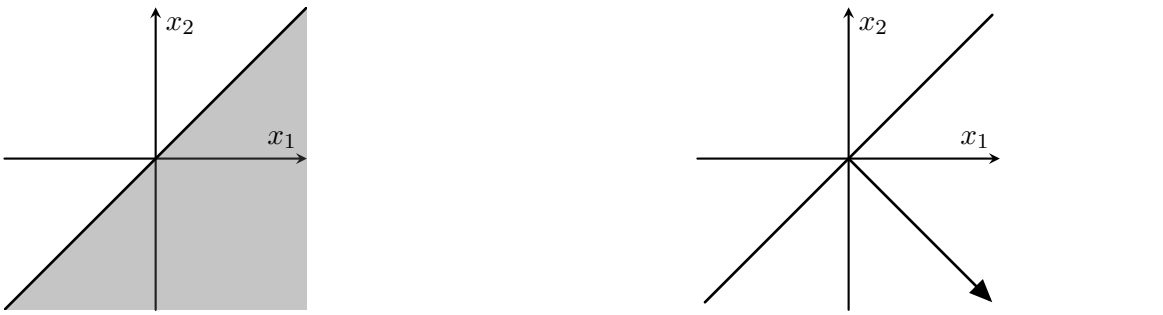

Figure 3.3: On the left hand side we have the polyhedral cone $C = \{x \in \mathbb{R}^2 \mid -x_1 + x_2 \leq 0\}$. The right hand side shows its decomposition into the lineality space and a pointed polyhedral cone. The lineality space is depicted by the straight line from bottom left to top right. The pointed polyhedral cone only consists of a single ray depicted by the arrow.

3.3 Double Description Method

The Minkowski-Weyl theorem for polyhedral cones has shown us, that it is possible to describe a polyhedral cone in two different ways. However, it is not immediately clear, how we could switch from one description to the other. In this section we present the *double description method*, which allows us to just that. It was first introduced by Motzkin et al. (1953) but independently rediscovered many times.

Definition 3.5 (DD pair; Fukuda and Prodon, 1995) Let $A \in \mathbb{R}^{m\times n}$ and $R \in \mathbb{R}^{n\times d}$ be real matrices. We say that (A, R) is a *DD pair*, if for all $x \in \mathbb{R}^n$

$$Ax \geq 0 \quad \iff \quad x = R\lambda \text{ for some } \lambda > 0$$

Note that the Minkowski-Weyl theorem for polyhedral cones tells us that such a pair always exists. From now on we will call the matrix A of a DD pair *representation matrix* and the matrix R the *generating matrix*. The polyhedral cone generated by them will be denoted by C.

Continuing forward we will consider the problem of finding a generating matrix R given a representation matrix A. Additionally, we will require R to be minimal, i.e. there exists no proper submatrix $R' \subsetneq R$ generating C. We did not choose this direction arbitrarely. For the problem considered later in this thesis, we are always given the representation matrix. We refer the interested reader to Farkas' Lemma (Farkas, 1902), which implies that (A, R) is a DD pair if and only if (R^T, A^T) is a DD pair.

In its simplest form, the double description method works as follows.[4] Given a real matrix $A \in \mathbb{R}^{m\times n}$, we select a subset $I \subseteq \{1, \ldots, m\}$ of row indices, such that we have DD pair (A_I, R), where A_I denotes the submatrix of A consisting of the rows indexed by I. Afterwards, we select a new row $i \in \{1, \ldots, m\} \setminus I$ and find a DD pair $(A_{I\cup\{i\}}, R')$ using our previously found DD pair (A_I, R). The last step is repeated until $A_I = A$, in which case we have found the desired DD pair (A, R).

[4] Since the method was first introduced by Motzkin et al. it has been optimized over and over again. Most notably by Fukuda and Prodon (Fukuda and Prodon, 1995).

Algorithm 1 Procedural form of the DD method

Input: A
Output: (A, R)
1: Obtain initial DD pair (A_I, R)
2: **while** $I \neq \{1, \ldots, m\}$ **do**
3: Choose $i \in I \setminus \{1, \ldots, m\}$
4: Construct $(A_{I\cup\{i\}}, R')$ from (A_I, R)
5: $I \leftarrow I \cup \{i\}$
6: $R \leftarrow R'$
7: **end while**

The method can essentially be split into an initialization step (line 1 of the procedure) and an iteration step (line 2 to 7). The easiest way to get an initial is to set $A = \emptyset$ and let R be a conic basis of $\mathbb{R}^n$.

The interesting part of the DD method is in the iteration step. A priori it is not clear how we can construct a new DD pair from a previously generated one. Consider an initial DD pair (A, R). Let A_i denote the newly chosen row. The constraint $A_i x \geq 0$ splits the space $\mathbb{R}^n$ into the following three separate spaces:

$$H^+ = \{x \in \mathbb{R}^n \mid A_i x > 0\}$$
$$H^0 = \{x \in \mathbb{R}^n \mid A_i x = 0\}$$
$$H^- = \{x \in \mathbb{R}^n \mid A_i x < 0\}$$

Remember that the column vectors $r_j \in R, j \in J$ are rays. Since we consider polyhedral cones, we know that these rays start at the origin and must therefore lie in one of the three spaces defined above. Thus we can use them to separate the index set J of our generating matrix R into three parts:

$$J^+ = \{j \in J : r_j \in H_i^+\}$$
$$J^0 = \{j \in J : r_j \in H_i^0\}$$
$$J^- = \{j \in J : r_j \in H_i^-\}$$

We call the rays indexed by J^+, J^0, J^- the *positive, zero* and *negative* rays with respect to i. The trick is to cleverly combine these rays to construct our new generating matrix R'. Clearly, the negative rays can no longer be part of a generating matrix. But if we would just use the old generating matrix without the negative rays we could loose important information. Therefore, we try to modify them such that they end up lying in the hyperplane

H^0. Geometrically, we copy every ray in H^- once for every ray in H^+. We then turn these copies into the direction of the corresponding ray in H^+ until they lie in the hyperplane H^0. These $|J^+| \cdot |J^-|$ new rays together with the $|J^+|$ positive and $|J^0|$ zero rays form our new generating matrix R'. This process is illustrated in Figure 3.4. The following lemma formally describes how to construct the additional rays and shows that the new pair $(A_{I\cup\{i\}}, R')$ is indeed a DD pair.

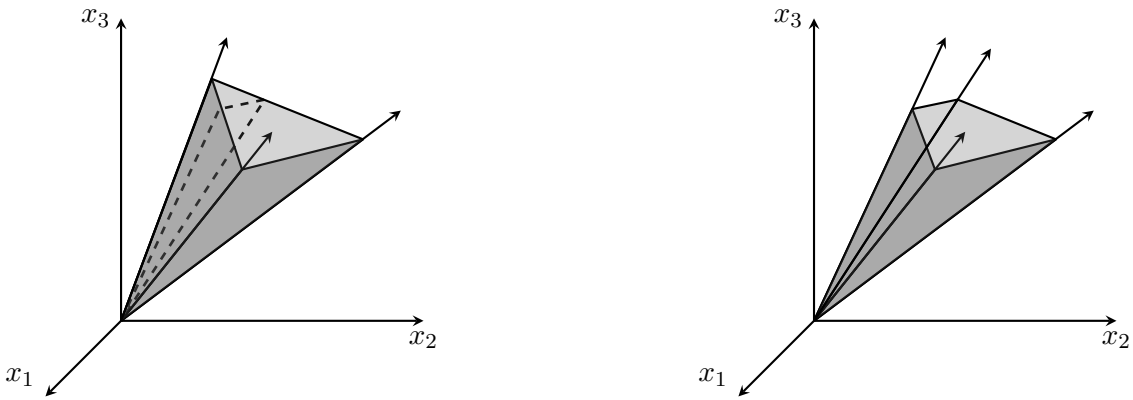

Figure 3.4: Illustration of an iteration step. The dashed area in the left figure shows the cut section by the newly introduced inequality. The right figure shows the result of the iteration step

Theorem 3.4 (Fukuda and Prodon, 1995) Let (A_I, R) be a DD pair and let i be a row index of A not in I. Then the pair $\left(A_{I\cup\{i\}}, R'\right)$ is a DD pair where R' is the $d \times |J'|$ matrix with column vectors r_j $(j \in J')$ defined by

$$J' = J^+ \cup J^0 \cup \left(J^+ \times J^-\right), \text{and}$$
$$r_{jj'} = (A_i r_j)\, r_{j'} - (A_i r_{j'})\, r_j \text{ for each } (j, j') \in J^+ \times J^-.$$

We still have not answered the question of what these generating rays are and if this generating matrix R is unique. It turns out that in the special case of pointed polyhedral cones the answers are quite interesting.

Theorem 3.5 (Fukuda and Prodon, 1995) Let C be a pointed polyhedral cone. Then, the minimal generating matrix R is unique up to scaling.

We will not formally prove this theorem. We will however give an intuitive explanation on why the minimal generating matrix R is unique. Note that jst applying the DD method

does not give us a minimal generating matrix R, since it generates a lot of *redundant* rays. Redundant rays of R are rays that could be omitted from R such that R would still generate the same polyhedron.

We already know that the generating matrix R consists of rays. In the case of pointed polyhedral cones these rays are *extreme rays*.

Definition 3.6 (Extreme Rays) A ray r of a pointed polyhedral cone $C \subseteq \mathbb{R}^n$ is called extreme ray, if it cannot be written as the conic combination of other rays $r_1, \ldots, r_k \in C, k \geq 2$.

If we would have a second minimal generating matrix $R' \neq R$, this would mean that there exists at least one extreme ray r in R that is not in R'. Since R' and R both generate the same cone C there must be rays in R' such that r is the conic combination of them. This is a contradiction. Therefore R must be unique. By the same argument we can see that every extreme ray of C must be in R.

3.4 Linear Programming

Unlike the name suggests, *linear programming* has nothing to do with computer programming. The term originated in the U.S. military, where proposed plans, strategies, and schedules were referred to as programs. George Dantzig, who worked for them on a planning problem, discovered that parts of these problems could be formulated as a system of linear inequalities. He therefore introduced the term *Programming in Linear Structure* before eventually changing it to linear programming (Dantzig and Thapa, 1997) . In essence, linear programming is a method to optimize a linear function where we have linear inequalities as constraints for the variables involved.

Definition 3.7 (Linear Program) Given numbers $n, m \in \mathbb{N}$, vectors $o \in \mathbb{R}^n, b \in \mathbb{R}^m$ and matrix $A \in \mathbb{R}^{n \times m}$, we define the linear program $\mathcal{L}$ as the problem of maximizing $o^T x$ subject to a system of linear inequalities $Ax \leq b$. A shorthand notation is given by $\mathcal{L} = \max\{o^T x \mid Ax \leq b\}$.

The linear inequalities for a linear program defined as above, are called *constraints*. We will use the term constraint and inequality interchangeably. We call $o^T x$ *objective function* and x satisfying the constraints is again called a *feasible solution*. Every x that also maximizes the objective function is called an *optimal solution*. If there exists no solution to the linear program we say that it is *infeasible*. If x can be arbitrarily large, we call the linear program *unbounded* and define $\max\{o^T x \mid Ax \leq b\}$ to be $-\infty$.

In *canonical form* a general linear program looks as follows

$$\begin{aligned} &\text{Maximize} \quad o^T x \quad \text{subject to} \\ &Ax \leq b \end{aligned}$$

We will again only consider linear programs where the set of constraints form a polyhedral cone, i.e. where $b = \mathbf{0}$. To get a better understanding of the topic at hand, we consider the following example of a linear program $\mathcal{L}$ depicted in Figure 3.5.

$$\begin{aligned} \text{Maximize} \quad -\frac{3}{4}x_1 - x_2 \quad &\text{subject to} \\ \frac{1}{2}x_1 - x_2 &\leq 0 \\ -3x_1 + x_2 &\leq 0 \end{aligned}$$

If not stated otherwise, we assume the variables to be real valued. In our simple example we can quickly realize that there does not exist an optimal solution, because we can let the objective function take on arbitrarily large values. In such a case the linear program is called *unbounded*.

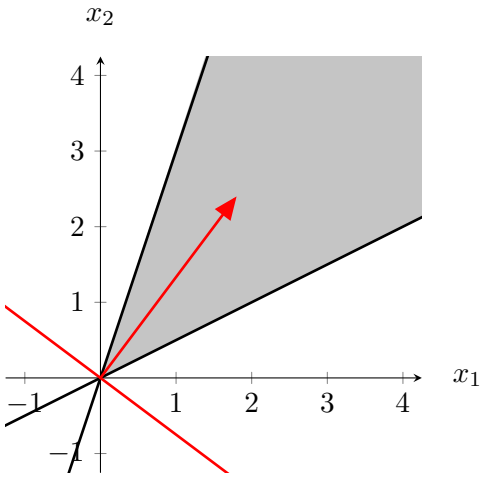

Figure 3.5: The feasible set is coloured in grey. The objective function is depicted by the red line. The arrow denotes the direction in which we can move the objective function to maximize it.

The most important concept in linear programming is *duality*.

Definition 3.8 (Dual of Linear Program) Given numbers $n, m \in \mathbb{N}$, vectors $o \in \mathbb{R}^n, b \in \mathbb{R}^m$

and matrix $A \in \mathbb{R}^{n \times m}$. The dual of the linear programming problem $\mathcal{L} \coloneqq \max\{o^T x \mid Ax \leq b\}$ is the problem $\mathcal{L}' \coloneqq \min\{b^T y \mid A^T y \geq o\}$.

The dual problem of a linear program is especially interesting because of the following theorem.

Theorem 3.6 (Strong Duality; e.g. Conforti et al., 2014) Let $\mathcal{L} \coloneqq \max\{o^T x \mid Ax \leq b\}$ and $\mathcal{L}' \coloneqq \min\{b^T y \mid A^T y \geq o\}$ for some numbers $n, m \in \mathbb{N}$, vectors $o \in \mathbb{R}^n, b \in \mathbb{R}^m$ and matrix $A \in \mathbb{R}^{n \times m}$. If $\mathcal{L}$ or $\mathcal{L}'$ is feasible and bounded, then both are feasible and bounded and their optimal solutions coincide.

3.5 Dantzig-Wolfe Decomposition and Column Generation

One of the most commonly used method to find optimal solutions of general linear programs is the Simplex algorithm (Dantzig et al., 1955). Still, for large linear programs the task of finding optimal solutions is computationally expensive. A method that works well on linear programs with a specific structure is the *Dantzig-Wolfe decomposition* (Dantzig and Wolfe, 1960). The method rewrites the original linear program and solves it using a technique called *column generation* (Ford Jr. and Fulkerson, 1958).

The Dantzig-Wolfe decomposition is defined for linear programs, where the constraint matrix can be written in the following form (the empty spaces denote zero-entries).

$$\begin{pmatrix} A_1 & A_2 & \cdots & A_n \\ B_1 & & & \\ & B_2 & & \\ & & \ddots & \\ & & & B_n \end{pmatrix}$$

Since we split the matrix into blocks, it also makes sense to split the coefficient vector o of the objective function and the variable vector x into corresponding sub-vectors o_i and

$x_i, 1 \leq i \leq n$. We rewrite the linear program as

$$\begin{aligned} \text{Maximize} \quad & \sum_{1 \leq i \leq n} o_i^T x_i \quad \text{subject to} \\ & \sum_{1 \leq i \leq n} A_i x_i \leq b_0 \qquad (3.7) \\ & B_i x_i \leq b_i \quad \text{for all } 1 \leq i \leq n \qquad (3.8) \end{aligned}$$

The constraints in (3.8) are independent of each other, i.e. they do not involve variables of other constraints in (3.8). We will refer to these constraints as *subproblem constraints*. If all constraints were of this form, we could solve the following linear program for all i, $1 \leq i \leq n$.

$$\begin{aligned} \text{Maximize} \quad & o_i^T x_i \quad \text{subject to} \\ & B_i x_i \leq b_i \end{aligned} \qquad (3.9)$$

The solutions to these linear programs could then be concatenated, leading to a solution of the original linear program by setting $x = (x_1, \ldots, x_n)^T, 1 \leq i \leq n$. The problem is that we also have constraints as in (3.7). These constraints (potentially) involve every variable x_i and are therefore called the *complicating constraints*.

In the problem considered in this thesis the vectors $b_i, 1 \leq i \leq n$ are set to 0. Hence we know that the solution for the subproblem constraints forms a polyhedral cone, i.e. $C_i \coloneqq \{B_i x_i \leq \mathbf{0}\}$ is a polyhedral cone. By the Minkowski-Weyl Theorem for Polyhedral Cones (Thm. 3.2) we know that every vector $v \in C_i$ can be written as the sum of finitely many generating rays $r_{i1}, \ldots, r_{1k}$, $k \geq 0$ of C_i. This means that we can replace the vectors x_i in the original linear program by a sum of generating rays. The rewritten form is called *master problem* and given below.

$$\begin{aligned} \text{Maximize} \quad & \sum_{1 \leq i \leq n} \sum_j o_i^T \lambda_{ij} r_{ij} \quad \text{subject to} \\ & \sum_{1 \leq i \leq n} \sum_j A_i \lambda_{ij} r_{ij} \leq b_0 \\ & \lambda_{ij} \geq 0 \quad \text{for all } i \text{ and } j \end{aligned}$$

Even though the master problem has less constraints, we still face a major challenge. The number of generating rays is usually too large, which makes finding a solution to the linear

program computationally infeasible.

This is where column generation comes into play. The basic idea of column generation is to first solve the master problem by only considering a small amount of the original variables involved. This smaller linear programming is called the *restricted master problem*. In the following example we coloured the entries in the restricted master problem in red. The black entries are currently not considered but part of the master problem.

$$\begin{pmatrix} a_{11} & \dots & a_{1k} & a_{1k+1} & \dots & a_{1n} \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{m1} & \dots & a_{mk} & a_{mk+1} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

After finding a solution for the restricted master problem, we try to check if there exists a variable that has not been considered yet, but improves the solution. If such a variable is found, we add it to our linear program. Adding a variable consists of adding a column to the matrix, which is where the name column generation originates from. The updated example where we added the variable x_{k+1} is given below.

$$\begin{pmatrix} a_{11} & \dots & a_{1k} & a_{1k+1} & \dots & a_{1n} \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{m1} & \dots & a_{mk} & a_{mk+1} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

After updating the restricted master problem with the new variable we solve it again. These steps are then repeated until we can show that there is no variable left that could improve our current optimal solution. The final solution is then the optimal solution for the original linear program.

To check if an improving variable exists, we solve the following linear program called the *pricing problem*.

$$\text{Minimize} \quad (A_i x_i)^T y - o_i^T x_i \quad \text{subject to}$$
$$B_i x_i \leq 0$$

The value y represents the solution of the dual linear program of the restricted master problem. The pricing problem determines an improving variable by checking if there exists a generating ray for the polyhedron generated by the inequalities such that the solution of the pricing problem is negative. If yes, the ray is added to the restricted master problem as a new column. If no ray produces a negative solution we are done.

Chapter 4

Cost Partitioning and the Dantzig-Wolfe Decomposition

Let us quickly recap what we have learned so far. We have seen that an optimal classical planning task can be solved using an admissible heuristic. Abstractions are admissible heuristics for which we can compute an optimal solution. Cost partitioning can then use the obtained solutions to construct an admissible heuristic for the original planning task.

The following sections show the linear program used for cost partitioning and how to apply the Dantzig-Wolfe decomposition to it. They closely follow the publication by Pommerening et al. (2021).

4.1 Restricted Master Problem

Let Π be our original planning task with induced state space $\Theta_\Pi = \langle \mathcal{S}, L, \text{cost}, \mathcal{T}, s_0, G \rangle$. Note that instead of a set of actions $\mathcal{A}$ we now talk about a set of labels L. These are the same objects but in this context using labels makes the notation easier. Katz and Domshlak (2010) have shown that given a set of abstractions $A = \{\alpha_1, \ldots, \alpha_n\}$, we can compute an optimal cost partitioning over A in the initial state by solving the following linear program.

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{1 \le i \le n} h_i && \text{subject to} \\
& \sum_{1 \le i \le n} c_{i\ell} \le \mathit{cost}(\ell) && \text{for all } \ell \in L \\
& d_{is_{0_i}} = 0 && \text{for all } \ell \in L \\
& d_{it} = d_{is} + c_{i\ell} && \text{for all } i \text{ and } \langle s, \ell, t \rangle \in T_i \\
& h_i \le d_{is^*} && \text{for all } i \text{ and } s^* \in G_i
\end{aligned}
$$

where the variables $d_i, 0 \le i \le n$, are called *distance variables* that represent the current

lower bound for the distance of state i to the nearest goal state. The variables $c_{i\ell}, 1 \leq i \leq n$, denotes the assigned cost for label $\ell \in L$ in the constraints of submatrix B_i of the Dantzig-Wolfe decomposition. This linear program has the desired structure to apply the Dantzig-Wolfe decomposition. The constraints in the first line of the linear program represent the complicating constraints. The remaining constraints form one subproblem for each abstraction $\alpha_i, 1 \leq i \leq n$. Remember that we have defined a matrix for Dantzig-Wolfe decomposition to be of the following form

$$D = \begin{pmatrix} \boxed{A_1} & \boxed{A_2} & \cdots\cdots & \boxed{A_n} \\ \boxed{B_1} & & & \\ & \boxed{B_2} & & \\ & & \ddots\ddots & \\ & & & \boxed{B_n} \end{pmatrix}$$

The submatrices A_i are defined as $A_i = (I \mid Z)$ where I is the identity matrix with size equal to the number of labels $\ell \in L$ and Z being the zero matrix with column size being the number of states in the abstraction plus an additional column for the heuristic value of the abstraction. The reason being that for every label $\ell \in L$ there exists a row in A_i and for every label, distance variable and the heuristic value there exists a column in A_i. The submatrices B_i have one row for every constraint of the subproblem. We therefore get $Dx \leq b$ as our system of inequalities, with

$$x = \left(\overbrace{c_{1\ell_1}, \ldots, c_{1\ell_{|L|}}, d_{1s_1}, \ldots, d_{1s_{|S_1|}}, h_1}^{\text{variables in submatrix } B_1}, \ldots, \overbrace{c_{n\ell_1}, \ldots, c_{n\ell_{|L|}}, d_{ns_1}, \ldots, d_{ns_{|S_n|}}, h_n}^{\text{variables used in submatrix } B_n} \right)^T$$

$$b = \left(\underbrace{cost(\ell_1), \ldots, cost(\ell_{|L|})}_{\text{result of } Ax}, \underbrace{0 \ldots, 0}_{\text{result of } Bx} \right)^T$$

where S_i denotes the set of states considered in the constraints of submatrix B_i and

$$A = \left(\boxed{A_1} \boxed{A_2} \ldots\ldots \boxed{A_n} \right) \qquad B = \begin{pmatrix} \boxed{B_1} & & & \\ & \boxed{B_2} & & \\ & & \ddots & \\ & & & \boxed{B_n} \end{pmatrix}$$

We know that the solution to a problem $B_i x \leq \mathbf{0}, 1 \leq i \leq n$, is given by the generating rays of its polyhedral cone. If we consider the rays r_{ij} of subproblem i they look as follows

$$r_{ij} = \left(c_{\ell_1}, \ldots, c_{\ell_{|L|}}, d_{i_1}, \ldots, d_{i_{|S_i|}}, h_i\right)$$

By comparing the master problem for the cost partitioning with the general master problem of the Dantzig-Wolfe decomposition, we realize that the coefficients λ_{ij} are all zero for the variables d_{ijs}. Therefore we can interpret a ray r_{ij} as a cost function for the labels in the abstraction together with the heuristic value h_i. Therefore we will sometimes refer to these rays as cost functions. The cost partitioning is then given by

$$cost_i(\ell) = \sum_j \lambda_{ij} c_{ij\ell}$$

where $c_{ij\ell}$ denotes the assigned cost of label $\ell in L$ in the generating ray r_{ij}. The heuristic value h_i is given by

$$h_i = \sum_j \lambda_{ij} h_{ij}$$

where h_{ij} denotes the value assigned to h_i in generating ray h_{ij}. Thus we can interpret the cost functions returned by the pricing problem as *candidate cost functions* with the heuristic value achieved under them. The restricted master problem then finds a linear combination for the returned candidate cost functions satisfying the cost partitioning. This cost partitioning is always optimal for the current set of candidate cost functions.

4.2 Pricing Problem

To compute the pricing problem we need the solution for the dual of the restricted master problem. Let y be this solution. The pricing problem for our subproblem i is given by

$$\begin{aligned}
\text{Minimize} \quad & \sum_{\ell \in L} y_\ell c_{i\ell} - h_i && \text{subject to} \\
d_{is_{0_i}} &= 0 && \text{for all } \ell \in L \\
d_{it} &= d_{is} + c_{i\ell} && \text{for all } i \text{ and } \langle s, \ell, t \rangle \in T_i \\
h_i &\leq d_{is^*} && \text{for all } i \text{ and } s^* \in G_i
\end{aligned}$$

Looking at the objective function it may seem odd that we are allowed to freely choose the costs for our labels, since we could just choose them to be negative. The second and third constraint penalize this strategy. If we assign negative costs to too many labels, we get a negative value for our heuristic h_i which in return makes our objective value larger.

This is where our thesis gets weird. In the following chapters we will consider methods that precompute **all** generating rays of our pricing problems. As already mentioned in the introduction this seems odd because the whole point of column generation is that we do not have to compute all solutions. However, we have reason to believe (Pommerening et al., 2021) that the number of solutions for the tasks we will consider are small and therefore the precomputation of all solutions of the pricing problem should take less time than repeatedly resolving the pricing and restricted master problem.

Chapter 5

Theoretical Analysis

This chapter contains the theoretical results derived during the study of the pricing problem. We first show, what happens to the linear constraints and the corresponding transition system if we us the Fourier-Motzkin elimination to remove all distance variables. We further propose a method to generate a system of constraints, whose feasible set is equivalent. In the second part we take a closer look at the result of the double description algorithm. We define which generating rays are interesting for our pricing problem and give an interpretation of what happens if we first decompose the cone into its lineality space and a pointed polyhedral cone.

5.1 Fourier-Motzkin and the Pricing Problem

Throughout this section we will refer to the following example of a pricing problem to better understand the introduced notation and results (we omit the objective function, since we are only interested in the generating rays).

$$
\begin{aligned}
d_0 &= 0 \\
d_0 &\leq d_0 + c_0 \\
d_0 &\leq d_2 + c_0 \\
d_1 &\leq d_0 + c_0 \\
d_1 &\leq d_0 + c_2 \\
d_2 &\leq d_0 + c_1 \\
d_2 &\leq d_1 + c_0 \\
h &\leq d_2
\end{aligned}
$$

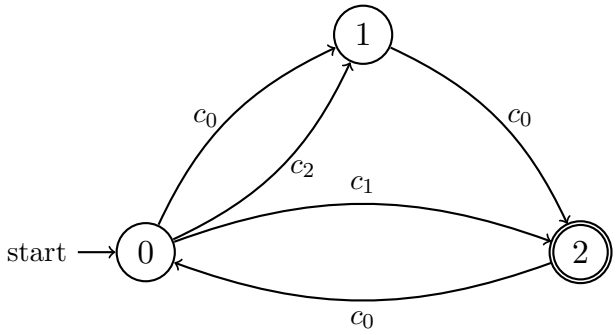

Figure 5.1: Graph corresponding to the linear program on the left.

The constraints of a general pricing problem are given by an induced abstract state space of the original planning task. For the sake of readability we omit the abstraction indices i.

$$\begin{aligned} d_0 &= 0 \\ d_t &\leq d_s + c_\ell \quad \text{for all } \langle s, \ell, t \rangle \in \mathcal{T} \\ h &\leq d_{s^*} \quad \text{for all goal states } s^* \in G \end{aligned}$$

Without loss of generality we can assume that only one goal state exists. If there are multiple goal states we can simply introduce a new goal state that has incoming edges from the original goal states where the label has zero cost.

Let $\Theta_\Pi = \langle \mathcal{S}, L, c, \mathcal{T}, s_0, s_* \rangle$ be an induced state space corresponding to one of our pricing problem defined as above. Let n be the number of states indexed from 0 to $n-1$ and let 0 denote the initial state. We call a sequence of transitions

$$\omega = (\langle s_0, \ell_0, s_1 \rangle, \ldots, \langle s_{m-1}, \ell_{m-1}, s_m \rangle), \ 1 \leq m \leq n-1$$

a *directed closed walk in* Θ_Π if $\langle s_i, \ell_i, s_{i+1} \rangle \in \mathcal{T}, 1 \leq i \leq m-1$ and $s_0 = s_m$. Note that we do not restrict the ℓ_i's to be unique. If $s_0 \neq s_m$ we call ω a *directed open walk from* s_0 *to* s_m *in* Θ_Π. The cost of a directed open / closed walk ω is given by the sum of costs of all labels used during the walk (if a label is used multiple times, the cost will be added every time). To denote the summed cost of a walk ω we will write $\sum_{\ell \in \omega} c_\ell$. An example for a directed open and closed walk with their summed cost can be seen in Figure 5.2.

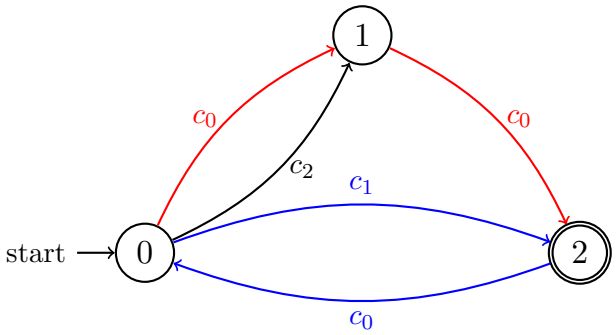

Figure 5.2: The directed open walk $\omega = (\langle 0, c_0, 1 \rangle, \langle 1, c_0, 2 \rangle)$ is coloured in red and has cost $c_0 + c_0$. The directed closed walk $\omega = (\langle 0, c_1, 2 \rangle, \langle 2, c_0, 0 \rangle)$ is coloured in blue and has cost $c_1 + c_0$.

We call a constraint *good* if it is in one of the following sets

$$I_{init} = \{d_0 = 0\} \tag{5.1}$$

$$I_{goal} = \{h \leq d_{s^*}\} \tag{5.2}$$

$$I_c = \{0 \leq \sum_{\ell \in \omega} c_\ell \mid \omega \text{ is a directed closed walk in } \Theta_\Pi\} \tag{5.3}$$

$$I_o = \{d_t \leq d_s + \sum_{\ell \in \omega} c_\ell \mid \omega \text{ is a directed open walk from } s \text{ to } t \text{ in } \Theta_\Pi, s, t \in \mathcal{S}\} \tag{5.4}$$

We call $Ax \leq \mathbf{0}$ a *good system of constraints* if all of its constraints are good.

Lemma 5.1 Let $Ax \leq \mathbf{0}$ be a good system of constraints. The Fourier-Motzkin elimination of a distance variable $d_i, 1 \leq i \leq n-1$ results in a good system of constraints.

Proof. For $n = 1$ the statement follows immediately, since we only have one state. Let $n \geq 2$, we want to eliminate $d_i, 1 \leq i \leq n-1$. The Fourier-Motzkin elimination retains all constraints that do not involve d_i. By assumption these constraints are already good. The remaining constraints all involve d_i and must be of the form

$$d_i \leq d_s + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } s \text{ to } i \tag{5.5}$$

$$d_t \leq d_i + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } i \text{ to } t \tag{5.6}$$

$$h \leq d_i + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } i \text{ to } s^* \tag{5.7}$$

Fourier-Motzkin elimination combines constraints (5.5) with (5.6) or (5.7). The first combination results in

$$0 \leq \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed closed walk, for } s = t$$

$$d_t \leq d_s + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } s \text{ to } t, \text{ for } s \neq t$$

Thus they are also good constraints. For the second possible combination we get

$$h \leq d_s + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } s \text{ to } s^*$$

which are also good constraints. This concludes the proof. □

Corollary 5.1 Let $Ax \leq \mathbf{0}$ be a good system of constraints. The Fourier-Motzkin elimination of all distance variables $d_i, 1 \leq i \leq n-1$ results in the following system of constraints

$$\begin{aligned} 0 &\leq \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed closed walk} \\ h &\leq \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from 0 to } s^* \end{aligned}$$

Proof. Iteratively applying Lemma 5.1 results in a system of the form

$$\begin{aligned} d_0 &= 0 \\ 0 &\leq \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed closed walk} \\ d_0 &\leq d_s + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } s \text{ to } 0 \\ h &\leq d_s + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from } s \text{ to } s^* \end{aligned}$$

Since every distance variable except d_0 has been eliminated, we are left with

$$\begin{aligned} d_0 &= 0 \\ 0 &\leq \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed closed walk} \\ h &\leq d_0 + \sum_{\ell \in \omega} c_\ell, \quad \omega \text{ is a directed open walk from 0 to } s^* \end{aligned}$$

Using the first constraint we can substitute d_0 by 0. This concludes the proof. □

We will refer to this system of constraints as A^*.

Fourier-Motzkin elimination produces a lot of redundant rays during each iteration. Efficiently identifying them is a problem on its own. It could therefore prove beneficial if we could somehow avoid this step or at least have less of them. Knowing the directed open and closed walks used in the constraints of A^* will allow us to build the system of constraints directly, without performing Fourier-Motzkin elimination.

Definition 5.1 (Simple Cycle in a State Space) Let Θ_Π be an induced state space of a planning task Π. A simple cycle $\omega = (\langle s_0, \ell_0, s_1 \rangle, \ldots, \langle s_{k-1}, \ell_{k-1}, s_k \rangle), 1 \leq k \leq n-1$ in Θ_Π is a directed closed walk with $s_i \neq s_j, 0 \leq i < j \leq k$ except for $s_0 = s_k$.

Definition 5.2 (Simple Path in a State Space) Let Θ_Π be an induced state space of a planning task Π. A simple path $\omega = (\langle s_0, \ell_0, s_1 \rangle, \ldots, \langle s_{k-1}, \ell_{k-1}, s_k \rangle), 1 \leq k \leq n-1$ in Θ_Π

is a directed open walk with $s_i \neq s_j, 0 \leq i < j \leq k$.

The directed open and closed walk in Figure 5.2 are a simple path and a simple cycle.

Lemma 5.2 Let ω be a simple cycle in Θ_Π. Then

$$0 \leq \sum_{\ell \in \omega} c_\ell$$

is a constraint of A^*.

Proof. Let $\omega = (\langle s_0, \ell_0, s_1 \rangle, \ldots, \langle s_{m-1}, \ell_{m-1}, s_m \rangle), 1 \leq m \leq n-1$ be a simple cycle in Θ_Π. Thus our original constraint system must contain the constraints

$$d_{i+1} \leq d_i + c_{\ell_i},\ 0 \leq i \leq m-1$$

Eliminating any of the distance variables $d_i, 0 \leq i \leq m-1$ results in a new system of constraints containing

$$\begin{aligned} d_{j+1} &\leq d_j + c_{\ell_j}, \text{ for } j \neq i \\ d_{i+1} &\leq d_{i-1} + c_{\ell_{i-1}} + c_{\ell_i} \end{aligned}$$

Iteratively eliminating all but two distance variables $d_i, d_j, 0 \leq i < j \leq m-1$ we get a system of constraints containing

$$\begin{aligned} d_i &\leq d_j + \sum_{0 \leq k \leq i-1} c_{\ell_k} + \sum_{j \leq k \leq m-1} c_{\ell_k} \\ d_j &\leq d_i + \sum_{i \leq k \leq j-1} c_{\ell_k} \end{aligned}$$

Eliminating either one of these variables results in

$$0 \leq \sum_{0 \leq k \leq m-1} c_{\ell_k}$$

This concludes the proof. □

Lemma 5.3 Let ω be a simple path in Θ_Π. Then

$$0 \leq \sum_{\ell \in \omega} c_\ell$$

is a constraint of A^*.

Proof. Analogous to the proof for simple cycles. □

Lemma 5.4 Let $\omega = (\langle s_0, \ell_0, s_1\rangle, \ldots, \langle s_{m-1}, \ell_{m-1}, s_m\rangle), 1 \leq m \leq n-1$ be a directed closed walk in Θ_Π. The constraint representing the walk can be written as the sum of constraints representing simple cycles in Θ_Π.

Proof. Let $\omega = (\langle s_0, \ell_0, s_1\rangle, \ldots, \langle s_{m-1}, \ell_{m-1}, s_m\rangle), 1 \leq m \leq n-1$ be a directed closed walk that is not a simple cycle (otherwise we are done). Since ω is not a simple cycle there exist states $s_i, s_j \in \mathcal{S}, 0 \leq i < j \leq n-1$ such that $s_i = s_j$. Let j be the smallest index such that this holds. Thus $\omega_1 = \Big(\langle s_i, c_{\ell_i}, s_{i+1}\rangle, \ldots, \langle s_{j-1}, c_{\ell_{j-1}}, s_j\rangle\Big)$ is a simple cycle in Θ_Π. Moreover,

$$\omega_2 = \Big(\langle s_0, c_{\ell_0}, s_1\rangle, \ldots, \langle s_{i-1}, c_{\ell_{i-1}}, s_j\rangle, \langle s_j, c_{\ell_j}, s_{j+1}\rangle, \ldots, \langle s_{m-1}, c_{\ell_{m-1}}, s_m\rangle\Big)$$

is a directed closed walk in Θ_Π. If ω_2 is a simple cycle we are done, since we can write the constraint corresponding to ω as the sum of the constraints corresponding to ω_1 and ω_2. If ω_2 is not a simple cycle we can repeat the procedure from before until we are left with a simple cycle. This must eventually happen, since we only consider finite state spaces. □

Lemma 5.5 Let $\omega = (\langle s_0, \ell_0, s_1\rangle, \ldots, \langle s_{m-1}, \ell_{m-1}, s_m\rangle), 1 \leq m \leq n-1$ be a directed open walk from the initial state 0 to the goal state s^* in Θ_Π, i.e. $s_m = s^*$. The constraint representing the walk can be written as the sum of a constraint representing a simple path and constraints representing simple cycles in Θ_Π.

Proof. Analogous to the proof for simple cycles. An example of such a decomposition is given in Figure 5.3. □

Definition 5.3 (Simple Constraint System) We call S^* the simple constraint system if its constraints are all constraints corresponding to a simple cycle or a simple path from 0 to h in Θ_Π.

Theorem 5.1 Let $\mathcal{F}_{A^*}$ and $\mathcal{F}_{\mathcal{S}^*}$ denote the feasible sets of the constraint systems A^* and S^*. We have

$$\mathcal{F}_{A^*} = \mathcal{F}_{S^*}$$

Proof. $\subseteq$: By Lemma 5.2 and 5.3 we have that every constraint in S^* is also a constraint in A^*. It follows that $\mathcal{F}_{A^*} \subseteq \mathcal{F}_{S^*}$.
$\supseteq$: By Corollary 5.1 every constraint in A^* corresponds to either a directed closed walk or a

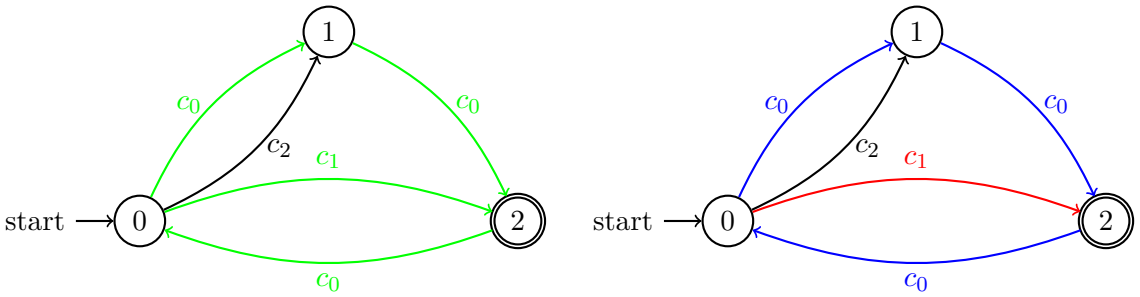

Figure 5.3: The directed open walk $\omega = (\langle 0, c_0, 1\rangle, \langle 1, c_0, 2\rangle, \langle 2, c_0, 0\rangle, \langle 0, c_1, 2\rangle)$ (green) can be seen on the left. Its decomposition into a directed simple path (red) and directed simple cycle (blue) is given on the right. This decomposition is not unique.

directed open walk from 0 to h in Θ_Π. By Lemma 5.4 and 5.5 we can write these constraints as the sum of constraints in S^*. Note that if a vector x satisfies the sum of two constraints it also satisfies the individual constraints. Hence we have that $\mathcal{F}_{A^*} \supseteq \mathcal{F}_{S^*}$ which concludes the proof. □

5.2 Double Description and the Pricing Problem

The double description gives us generating rays for the polyhedral cone forming our solution space. Still not all of them generate a new column. Remember that a ray generates a new column if it produces a negative objective value in the pricing problem. For this to happen, either we must assign a negative cost to at least one of the labels or the heuristic value is larger than zero. We will call a ray that satisfies this *interesting*. Throughout this section we assume every ray r to be of form $(c_1, \ldots, c_m, h)$ where m denotes the number of labels. Moreover, we call the rays $(0, \ldots, 0, -1), (1, 0 \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, \ldots, 1, 0)$ *basic*.

Proposition 5.1 Let C be the solution space of our system of constraints. The basic rays are always inside the polyhedral cone C.

Proof. Remember that the pricing problem consists of the following constraints:

$$0 \leq \sum_{\ell \in \omega_1} c_\ell$$
$$h \leq \sum_{\ell \in \omega_2} c_\ell$$

where ω_1 is a simple cycle and ω_2 is a simple path from the initial state to the goal state of

the considered state space. The rays from the statement all satisfy these constraints. □

This statement also makes intuitive sense, as it tells us that we can always increase the cost of any label or decrease the heuristic value.

Corollary 5.2 Let $C \subseteq \mathbb{R}^n$ be the pointed polyhedral cone generated by the rays in Proposition 5.1. For any polyhedral cone $Q \subseteq \mathbb{R}^n$ with generating rays $r_1 \dots, r_k$ we have

$$r_i \text{ is interesting} \iff r_i \notin C$$

Proof. $\implies$: From Proposition 5.1 we know that C does not contain any interesting rays. $\impliedby$: Since $r_i \notin C$, it must either have a negative cost for a label or the heuristic value is larger than 0. This is exactly the definition of an interesting ray. □

Proposition 5.2 If the initial state is not a goal state and the lineality space is nonempty, the lineality space contains only interesting cost functions.

Proof. Assume for contradiction that $r \in \text{lin}(C)$ is not interesting. Since $r \neq \mathbf{0}$ we must either have at least one label with cost greater than 0 or the heuristic value must be negative. If $h < 0$, there must exist a simple path with total negative costs. Therefore at least one of the costs for a label must be negative, Thus r is interesting which is a contradiction. If $h = 0$, there must exist a positive cost for a label. Since we only consider alive graphs, this label must be part of either a simple cycle or a simple path. Thus, there must be at least one label that compensates for its positive cost, i.e. there exists a label with negative costs. Thus r is interesting which is again a contradiction. Therefore r must be interesting. □

Proposition 5.3 Let C be a polyhedral cone. Then the following statements are equivalent

(i) $r_i \in \text{lin}(C)$

(ii) The ray r_i achieves the same heuristic value for all simple paths from the initial to the goal state, i.e. all simple paths from the initial to the goal state have the same cost. All simple cycles have cost 0.

Proof. Since r_i lies in the lineality space we achieve equality for every constraint. The statement follows immediately. □

A second important property a ray can have is called saturated (Seipp et al., 2020). For such a ray the cost of each label can no longer be reduced without affecting the heuristic value. We are especially interested in saturated rays, since they make our constraints tighter.

Proposition 5.4 Let $C = \{Ax \leq \mathbf{0}\}$ be a pointed polyhedral cone and let R be such that (A, R) is a DD-pair. Then every extreme ray $r \in C$ is saturated.

Proof. Assume for contradiction that $r \in C$ is an extreme ray that is not saturated. Thus we can either increase h or decrease the cost for a label without changing the rest of the entries. For the first case we get a new vector $c' = r + (0, \ldots, 0, \lambda)$ for $\lambda > 0$. Thus $r = c' + \lambda(0, \ldots, 0, -1)$ which by Proposition 5.1 is a conic combination of rays inside C. This is a contradiction to the fact that r is an extreme ray. For the second case let us assume that we decrease the cost of the first label. We get a new vector $c' = r + (-\lambda, 0, \ldots, 0)$ for $\lambda > 0$. Thus $r = c' + \lambda(1, 0, \ldots, 0)$ which by Proposition 5.1 is a conic combination of rays inside C. This is a contradiction to the fact that r is an extreme ray and we are done. □

Conjecture 1 (Raphaels Conjecture) The converse of Proposition 5.4 also holds, that is every saturated ray r of a pointed polyhedral cone C is an extreme ray.

Chapter 6

Experiments

We used the theory presented in the previous chapters to design five different algorithms to find the generating rays of our pricing problems. This chapter presents these algorithms and the results we got when evaluating them on planning tasks.

6.1 Algorithms

We pursue two different approaches. The first approach uses the double description method on the original pricing problem to compute the generating rays. Afterwards we us a projection to eliminate the distance variables from our generating rays.

For the second approach we first use a projection to eliminate the distance variables. Afterwards we use the double description method to compute the generating rays.

The two approaches results in five different algorithms.

1. DDProj: This algorithm first uses the double description method on the original pricing problem. The projection consists of simply dropping the distance variables from the rays and removing duplicate rays. Note that for this approach we (almost) always end up with redundant generating rays. This is not the case in the other approach.

2. FMDD: For the projection we use Fourier-Motzkin elimination. The double description method then computes the generating rays.

3. FMDecDD: We again use Fourier-Motzkin elimination to project out the distance variables in the constraints of the pricing problem. Afterwards we compute the lineality space for our new constraints and use the decomposition method presented in Theorem 3.3 to construct a pointed polyhedral cone. We compute the generating rays of the pointed polyhedral cone with the double description method and combine

it with the generating rays of the lineality space to get the generating rays for our original problem.

4. SCDD: This algorithm is the same as FMDD except that we calculate the simple constraint system instead of using Fourier-Motzkin elimination.

5. SCDecDD: Analogous to the FMDecDD algorithm but we eliminate the distance variables by constructing the simple constraint system.

6.2 Setup

The algorithms were implemented in Python 3.8. Fourier-Motzkin elimination was implemented as presented in chapter 3 and we only removed (redundant) constraints if they were not unique. We used the double description method of the pycdd library[1] which is based on the cdd library by Fukuda programmed in C[2]. The directed simple paths and cycles were computed using the python version of the igraph library[3]. We obtained the nullspace by using the module scipy[4].

Our benchmark consisted of the abstractions in two (SYS2) and three (SYS3) variables for the 1827 tasks without conditional effects from the optimal sequential tracks of the International Planning competitions 1998-2018[5]. However, we were only able to use 1590 of these tasks, since for the remaining 237 tasks we were not able to generate all abstractions.

We ran the experiment on Intel Xeron Silver 4114 processors running on 2.2 GHz at sciCORE scientific computing cluster of the University of Basel[6]. We set the time limit to 5 minutes and limited the memory to 2 GiB per task.

6.3 Results

6.3.1 Solved Tasks

In a first step we compared the number of tasks we were able to solve using the different algorithms. The results are given in Table 6.1. As expected the number of tasks we were able to solve was much higher in Sys2 than Sys3. For Sys2 we were able to solve 1372

[1] https://pypi.org/project/pycddlib/
[2] https://github.com/cddlib/cddlib
[3] https://igraph.org/python/
[4] https://scipy.org
[5] http://ipc.icaps-conference.org
[6] https://www.scicore.unibas.ch

out of the 1590 tasks with all algorithms, whereas for Sys3 this was only the case for 138 tasks. It is also evident that the approach of getting rid of the distance variables with either Fourier-Motzkin elimination or constructing the simple constraint system outperforms the method of directly applying the double description method, since we removed the distance variables. The algorithms that construct the simple constraint system were even able to solve all tasks in Sys2.

Sys2	Solved	Time Limit Reached	Memory Limit Reached
DDProj	1398	**192**	–
FMDD	1560	9	**21**
FMDecDD	1548	12	**30**
SCDD	**1590**	–	–
SCDecDD	**1590**	–	–

Sys3	Solved	Time Limit Reached	Memory Limit Reached
DDProj	163	**1427**	–
FMDD	196	327	**1107**
FMDecDD	192	274	**1124**
SCDD	351	**1013**	226
SCDecDD	**356**	**915**	319

Table 6.1: Number of tasks solved per algorithm and reason why the algorithm was not able to solve a task. Highest number of solved tasks and main reason for not completing remaining tasks are marked bold.

Considering the source of not completing the tasks we can see that the DDProj never reaches the memory limit even though it is the algorithm that fails to solve the task the most. This is probably due to the fact, that the pycdd library removes redundancy during runtime. We will take a closer look at redundant constraints in a moment.

Summarizing we can state that the bottleneck for the algorithms DDProj, SCDD and SCDecDD is given by the time limit. FMDD and FMDecDD seem to reach the memory limit more often before running out of time. They still solve more tasks than DDProj.

6.3.2 Total Runtime

In a next step we compared the runtime (in s) for the tasks that were solved by all algorithms. For this we calculated the mean and the median of the runtime for all tasks that were solved

by all five algorithms. The results are presented in Table 6.3.

Sys2	Mean Total Runtime	Median Total Runtime
DDProj	5.593	0.392
FMDD	**4.226**	**0.126**
FMDecDD	8.591	0.166
SCDD	4.727	0.156
SCDecDD	9.301	0.195

Sys3	Mean Total Runtime	Median Total Runtime
DDProj	7.865	0.137
FMDD	0.988	0.147
FMDecDD	1.455	0.200
SCDD	**0.202**	**0.104**
SCDecDD	0.307	0.136

Table 6.2: Measurements of total runtime (in s) for tasks that were solved by all five algorithms. Fastest runtimes are bold.

The results show that there is a significant difference between the mean and median runtime for tasks in Sys2. This difference suggests that if we were able to solve a task, we were usually fast in doing so and there were only few outliers. However, these outliers still seemed to significantly impact the mean of the total runtime. This assumption is further strengthened by Figure 6.1 which shows that most of the solved tasks in Sys2 have a runtime below one second and that there exist some extreme outliers with runtimes of over 100 seconds. For tasks in Sys3 the proportion of tasks solved in under one second seems smaller but there still exists more tasks for which the runtime is below one second.

To get an indicator for the runtime of an algorithm, we tried to measure the complexity of a task by multiplying the number of constraints of the original problem by the number of variables involved. We would assume that the total runtime for a task increases with its complexity. Figure 6.1 seems to confirm our assumption for most tasks but there seems to be a group of tasks in Sys3 that have a low complexity but still produce a high runtime (bottom right of the right figure in Figure 6.1). This also destroys our hope that the generating rays for the pointed polyhedral cone constructed during the decomposition might be easier to compute using the double description method.

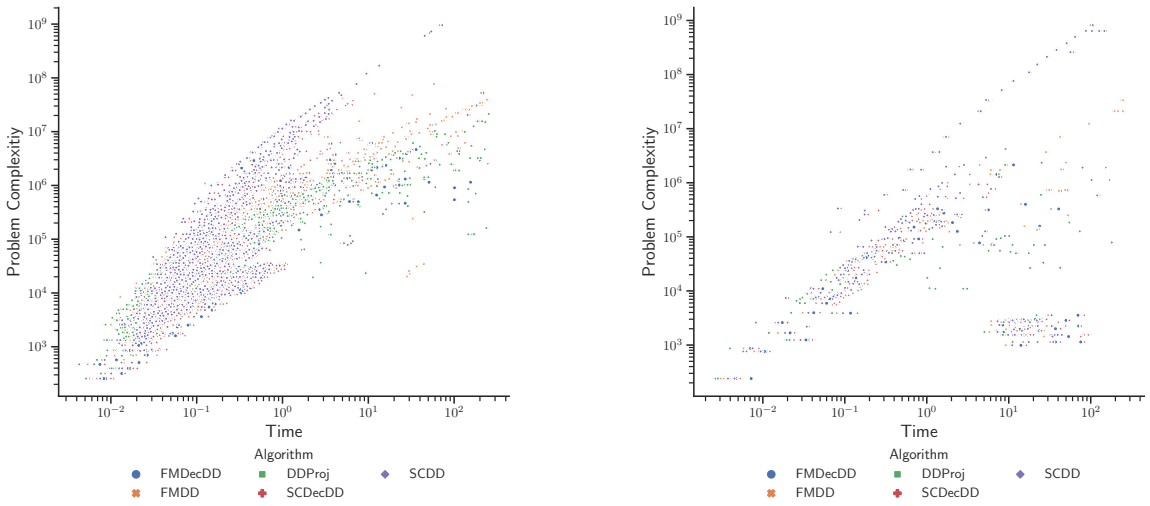

Figure 6.1: Total runtime versus complexity of the problem for tasks in Sys2 (left) and Sys3 (right) that were solved by all five algorithms.

Sys2	Mean Total Runtime	Median Total Runtime
DDProj	5.593	0.392
FMDD	**4.226**	**0.126**
FMDecDD	8.591	0.166
SCDD	4.727	0.156
SCDecDD	9.301	0.195

Sys3	Mean Total Runtime	Median Total Runtime
DDProj	7.865	0.137
FMDD	0.988	0.147
FMDecDD	1.455	0.200
SCDD	**0.202**	**0.104**
SCDecDD	0.307	0.136

Table 6.3: Measurements of total runtime (in s) for tasks that were solved by all five algorithms. Fastest runtimes are bold.

Although the FMDD and SCDD algorithms seem to have a similar performance regarding runtime, we can see that this is not the case when considering the 1590 (Sys2) and 194 tasks (Sys3) that could be solved by both of them. Table 6.4 shows the difference in total runtime while also considering the time used by the double description method after the projections. The table clearly indicates that the SCDD algorithm is superior with respect

to the runtime and that the runtime of the double description method is negligible when compared to the time used for the projection.

Sys2	Mean Total Runtime	Median Total Runtime	Mean DD Runtime	Median DD Runtime
FMDD	7.597	0.037	0.192	0.002
SCDD	0.387	0.036	0.123	0.002

Sys3	Mean Total Runtime	Median Total Runtime	Mean DD Runtime	Median DD Runtime
FMDD	19.944	0.262	0.682	0.011
SCDD	3.982	0.066	0.304	0.008

Table 6.4: Measurements of total runtime and runtime of the double description method (in s) for tasks that were solved by FMDD and SCDD.

6.3.3 Peak Memory Consumption

Besides measuring the runtime for the tasks we also measured peak memory consumption (in MiB). The results can be seen in Table 6.5.

Sys2	Mean Peak Memory Consumption	Median Peak Memory Consumption
DDProj	135.8	147.7
FMDD	**132.0**	**147.0**
FMDecDD	135.5	148.6
SCDD	**132.0**	147.2
SCDecDD	135.8	148.9

Sys3	Mean Peak Memory Consumption	Median Peak Memory Consumption
DDProj	153.8	**147.1**
FMDD	140.3	147.2
FMDecDD	144.6	148.7
SCDD	**138.7**	147.3
SCDecDD	143.6	148.2

Table 6.5: Measurements of peak memory consumption for tasks that were solved by all five algorithms. Lowest values are marked bold.

Interestingly enough, the peak memory consumption for Sys2 and Sys3 and all five algorithms is approximately the same. At first, this seems to indicate that the double description

method could be responsible for the peak memory, since it is the only component all five algorithms have in common. What seems odd with this explanation is that we never reach the memory limit when using the double description method before projecting but we reach the memory limit a lot when first using the projection for tasks in Sys3. This would imply that either the input matrix for the double description method is larger after applying the projection and therefore the double description method is able to reach the memory limit or the memory limit is reached during the projection. Figure 6.2 suggests that there is little correlation between the complexity and the peak memory. This suggests that the memory limit is indeed reached during projection.

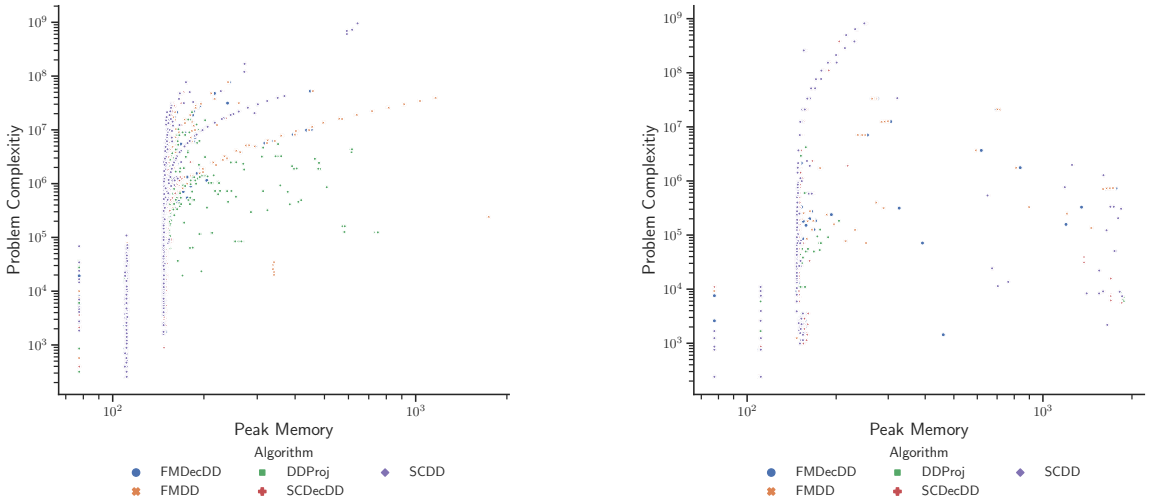

Figure 6.2: Peak memory consumption versus complexity of the problem for tasks in Sys2 (left) and Sys3 (right) that could be solved by all five algorithms.

Summarizing we can conclude that our definition of complexity seems to be a good indicator for the runtime in most cases. Moreover, we observed that the algorithms involving the simple constraints approach seem to outperform the others with respect to runtime for tasks in Sys3 and when the problems have a higher complexity. For the peak memory performance we observed that all five algorithms use a similar amount of peak memory.

6.3.4 Redundant Constraints After Projection

Fourier-Motzkin elimination could potentially result in an exponential number of constraints compared to the original number of constraints. The same is true for the number of constraints in the simple constraint system, since there can be exponentially many simple paths

and cycles in a graph. Table 6.3 shows the number of constraints existing before applying the double description method, that is after projecting out the distance variables with Fourier-Motzkin elimination or with the simple constraint system. Recall that for tasks in Sys2 the number of times we reached the memory limit was almost identical for both type of projections. The difference was large for tasks in Sys3. For these tasks the number of constraints after the projection is higher when using Fourier-Motzkin elimination. This suggests that there are more redundant constraints generated which could explain why the memory limit was reached far more often and less tasks could be solved.

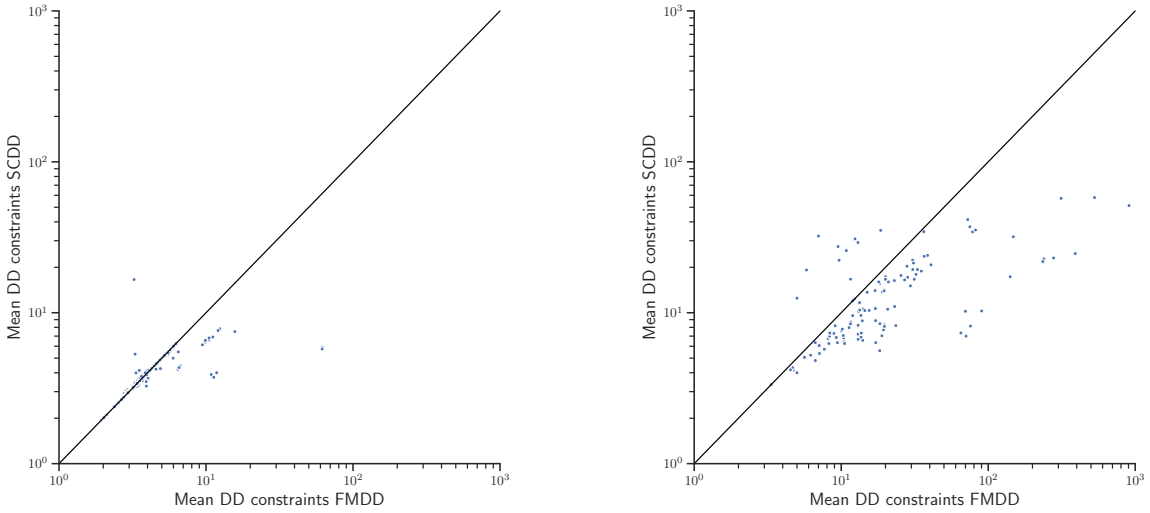

Figure 6.3: Number of constraints for the matrices in Sys2 (left) and Sys3 (right) before applying the double description method.

Chapter 7

Conclusion

7.1 Discussion of Results

The aim of this thesis was to empirically compare different approaches that compute the generating rays of the pricing problem. We have shown that the approach where we first project and then apply the double description method is superior to the one where we directly use the double description method and then project, with respect to runtime and numbers of tasks solved. We have found an indicator for the runtime of our algorithms. However, this indicator fails when we try to use it to predict peak memory consumption. Moreover, we have presented a method that replaces the Fourier-Motzkin elimination by constructing a system of constraints using the simple paths and simple cycles of the transition system of the planning task. This approach was able to solve more tasks and had a better runtime than the approach using Fourier-Motzkin elimination and created less redundant constraints. Furthermore we showed that decomposing the solution space into a pointed polyhedral cone and lineality space, we additionally get an interpretation for the solutions that lie in the respective object. The use of the decomposition did not result in a faster runtime and was not able to solve significantly more tasks for both projection methods.

7.2 Future Work

The reason we completed this thesis, was the work by Pommerening et al. (2021) where they used the Dantzig-Wolfe decomposition to solve planning tasks faster. It would be interesting to test if precomputing the generating rays of the pricing problems further improves the performance of their approach.

Not all generating rays of the pricing problems result in an added variable for the original linear program. We know that there are rays that will never produce a new column (e.g.

the basic rays in chapter 5.2). It would be nice, if we could already dismiss them while calculating the generating rays to our pricing problems.

Removing the distance variables of the pricing problem using either projection method creates a lot of redundant constraints and increases the complexity. We could probably significantly improve the runtime and peak memory consumption of our algorithms if we would be able to remove the redundant constraints during the projection.

Bibliography

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–655, 1995.

Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer Programming*. Springer, 2014.

George B Dantzig and Mukund N Thapa. *Linear Programming, 1: Introduction*. Springer, 1997.

George B Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations research*, 8:101–111, 1960.

George B Dantzig, Alex Orden, and Philip Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5:183–195, 1955.

Julius Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902:1–27, 1902.

Lester R Ford Jr. and Delbert R Fulkerson. A Suggested Computation for Maximal Multi-Commodity Network Flows. *Management Science*, 5:97–101, 1958.

Jean B J Fourier. Solution d'une question particuliere du calcul des inégalités. *Nouveau Bulletin des Sciences, par la Société philomatique de Paris*, 99:100, 1826.

Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, pages 91–111. Springer, 1995.

Robert C Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, 170: 1123–1136, 2006.

Michael Katz and Carmel Domshlak. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, 174:767–798, 2010.

Drew M McDermott. The 1998 AI planning systems competition. *AI magazine*, 21:35–35, 2000.

Hermann Minkowski. *Geometrie der Zahlen*. BG Teubner, 1910.

Theodore S Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Azriel Press, 1936.

Theodore S Motzkin, Howard Raiffa, Gerald L Thompson, and Robert M Thrall. The Double Description Method. *Contributions to the Theory of Games*, 2:51–73, 1953.

Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From Non-Negative to General Operator Cost Partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, page 3335–3341. AAAI Press, 2015.

Florian Pommerening, Thomas Keller, Valentina Halasi, Jendrik Seipp, Silvan Sievers, and Malte Helmert. Dantzig-Wolfe Decomposition for Cost Partitioning. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling*, pages 271–280, 2021.

Jendrik Seipp, Thomas Keller, and Malte Helmert. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67:129–167, 2020.

Hermann Weyl. The Elementary Theory of Convex Polyhedra. *Contributions to the Theory of Games*, 1:3–18, 1949.