JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

IMPROVING INTERPOLANTS OF NON-CONVEX POLYHEDRA WITH LINEAR ARITHMETIC AND PROBABLY APPROXIMATELY CORRECT LEARNING FOR BOUNDED LINEAR ARRANGEMENTS

Dissertation
zur Erlangung des Grades

"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

BJÖRN BEBER

geboren in Berlin

Mainz, den 26.02.2018

Oral exam date: 16.05.2018

ABSTRACT

This thesis is split into two parts. In the first part on the top level is a model-checking algorithm, which verifies safety properties of a linear hybrid automaton. During computation it has to handle intermediate state sets, which are described as non-convex polyhedra, i.e. subsets of $\mathbb{R}^n \times \mathbb{B}^m$ with linear hyperplanes as boundaries. The computation of this algorithm had problems handling the huge amount of linear constraints in the description of those state sets. Therefore we compute a Craig interpolant of the state set Φ and the complement of an epsilon-bloated state set $\Phi' = (\Phi \cup \Phi_\varepsilon)^C$, with a preferably lower amount of linear constraints.
We propose an algorithm that reduces the number of constraints of a given interpolant for the pair (Φ, Φ') of state sets. The approach is to select two linear constraints of the current interpolant and, if successful, substitute it by only one new linear constraint.
Additionally, we propose two applications of this algorithm in similar context. With the first application we construct lower bounds for the problem above, in the specific case that Φ and Φ' share the same boundary. As a second application we propose an algorithm to reduce the number of bits used in the description of a state set, while limiting the overapproximation by taking a superset of the given state set into account.
Our results show that the quality of our main algorithm is superior to all related algorithms. In addition it is even faster than all related algorithms, when we do not count the algorithm that only removes redundant linear constraints.
In the second part we switch to the context of theoretical machine learning. More specifically we propose an algorithm that is a Probably Approximately Correct (PAC) learner algorithm for the concept class of bounded linear arrangements in fixed dimension. Furthermore we introduce an Integer Linear Program (ILP) that models this problem, which then is solved by a Column Generation (CG) approach to handle the large amount of variables used in the ILP. Finally we speed up the algorithm by a randomized approach. Instead of taking all samples into account at once, we build up a subset of samples that consists of significant samples.
Additionally to the theoretical results, our evaluation confirms that our final approach can handle more reasonable datasets.

Diese Arbeit ist in zwei Abschnitte unterteilt. Im ersten Teil ist unser Problem angesiedelt als Anwendung für einen Model-Checking-Algorithmus, der sicherheitskritische Eigenschaften eines linearen hybriden Automaten verifiziert. Für die Berechnung werden Zustandsmengen als nicht konvexe Polyeder beschrieben, d. h. Mengen aus $\mathbb{R}^n \times \mathbb{B}^m$ dessen Grenzen durch lineare Hyperebenen beschrieben werden. Die Berechnungszeit des Algorithmus scheitert an zu großen Mengen von linearen Hyperebenen in der Beschreibung der Zustandsmengen. Wir suchen eine Craig Interpolante für eine Zustandsmenge Φ und dem Komplement der um Epsilon erweiterten Zustandsmenge $\Phi' = (\Phi \cup \Phi_\varepsilon)^C$ mit einer möglichst geringen Anzahl an linearen Ungleichungen.
Wir stellen einen Ansatz vor, der die Anzahl an linearen Ungleichungen einer gegebenen Interpolante für ein Paar von Zustandsmengen (Φ, Φ') reduziert. Unser Ansatz wählt zwei lineare Ungleichungen aus der momentanen Interpolante aus und versucht diese durch eine neu berechnete Ungleichung zu ersetzen. Zusätzlich stellen wir zwei Anwendungen unseres Algorithmus in dem Kontext des Model-Checking-Algorithmus vor. Als erste Anwendung berechnen wir untere Schranken für unser Problem in dem Fall, dass sich beide Zustandsmengen Φ und Φ' eine gemeinsame Grenze teilen. Die zweite Anwendung reduziert die Anzahl der verwendeten Bits in der Beschreibung einer Zustandsmenge unter Berücksichtigung einer Obermenge.
Durch unsere Analyse zeigen wir, dass die Qualität unseres Hauptalgorithmus besser ist, als die verwandter Methoden. Zusätzlich ist unsere Methode schneller als alle verwandten Methoden, wenn man die Methode vernachlässigt, die nur Redundanzen beseitigt.
Im zweiten Teil beschäftigen wir uns mit dem Thema des theoretischen maschinellen Lernens, insbesondere mit dem *Wahrscheinlich Annähernd Richtigen Lernen* (PAC). Wir stellen einen PAC-Lerner für beschränkte lineare Anordnungen in fixierter Dimension vor. Zudem modellieren wir ein *Ganzzahliges Lineares Programm* (ILP) und lösen dieses mit *Spaltengenerierung* (CG), um mit der großen Menge an Variablen umgehen zu können. Als letzte ausschlaggebende Verbesserung stellen wir einen randomisierten Ansatz vor. Statt alle Proben gleichzeitig zu betrachten, baut der randomisierte Ansatz eine Teilmenge von charakteristischen Proben auf.
Zusätzlich zu den theoretischen Resultaten zeigt die Analyse, dass unser endgültiger Algorithmus deutlich komplexere Benchmarks berechnen kann, als der Basis-Algorithmus.

DANKSAGUNG

An erster Stelle möchte ich meinem Doktorvater für die gemeinsame Zeit, das offene Ohr und viele hilfreiche Hinweise während meiner Arbeit als Doktrand danken.

Mein weiterer Dank geht an meinen zweiten Gutachter für seine sehr spontane Zusage Gutachter für meine Dissertation zu werden.

Ein weiterer Dank geht an alle Mitglieder des Instituts für Informatik an der Johannes Gutenberg-Universität in Mainz, für viele interessante Diskussionen, guten wie auch schlechten Kaffee, und immer einer netten Athmospähre untereinander.

Ich danke zudem allen Mitgliedern des Sonderforschungsbereichs "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, [7]), vorallem den Mitgliedern der FOMC Projektgruppe für die Einführung in das Thema "Model-Checking", einen anregende Austausch und einer guten Zusammenarbeit.

Sehr dankbar bin ich all meinen Kollegen und Freunden, die mir in vielfacher Hinsicht geholfen haben.

Mein größter Dank geht an meine Frau, die immer hinter mir steht und mich in jeder Hinsicht unterstützt und an mich glaubt. Zuletzt möchte ich noch meiner gesamten Famile für alles danken was sie für mich getan haben. Ein spezieller Dank geht an meine Mutter für die Unterstützung meines Studiums und ihrem Glauben an mich.

Björn Beber Mainz, den 26.02.2018

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AIG And-Inverter-Graph

BILP Binary Integer Linear Program

CEGAR Counterexample-Guided Abstraction Refinement

CG Column Generation

CNF Conjunctive Normal Form

DC Don't Care Set

DNF Disjunctive Normal Form

ILP Integer Linear Program

KNN k-Nearest Neighbor

LHA+D Linear Hybrid Automaton Extended with Discrete States

LINAIG Linear-And-Inverter-Graph

LP Linear Program

MP Master Problem

NRC Non-Redundancy-Certificate

PAC Probably Approximately Correct

PP Pricing Problem

RMP Restricted Master Problem

SAT Boolean Satisfiability Problem

SMT Satisfiability Modulo Theory

SVM Support Vector Machine

Part I

IMPROVING INTERPOLANTS WITH LINEAR ARITHMETIC

1 INTRODUCTION

The combination of compact descriptions and Craig interpolants for the Boolean Satisfiability Problem (SAT) and the problem of SMT came into focus during the last years, mainly in form of applications in formal verification. There interpolants are usually computed based on resolution proofs of unsatisfiability for the conjunction of two formulas and were used as overapproximations of reachable state sets. Due to the fact that an interpolant is by far not unique, this lead to the research of computing interpolants that are beneficial in other ways.
In our case the problem arose in the context of an algorithm based on Counterexample-Guided Abstraction Refinement (CEGAR) for the verification of safety properties of a linear hybrid automaton. In this context we propose an algorithm that computes interpolants for two non-convex polyhedra. These non-convex polyhedra represent sets of states in this linear hybrid system. Typically these hybrid automata have a large discrete state space. Therefore these non-convex polyhedra can be described by a SMT formula, with respect to linear arithmetic as the underlying theory. The safety properties are given by an initial state set and an unsafe state set, where the automaton is considered to be safe if there exists no trajectory from the initial state set to the unsafe state set. This is done by a reachability analysis. In our case the algorithm enlarges the unsafe state set stepwise by adding all points that lead to an unsafe state within one step. A crucial step in this computation is the elimination of quantifiers, which highly depends on the number of linear constraints used in the description of the intermediate state set. Therefore we search for an overapproximation of the current state set that can reach the unsafe state set by an interpolant. In our case we try to minimize the number of linear constraints used in the description of the interpolant, where the interpolant can be chosen over all interpolants of two non-convex polyhedra. Since the overlying problem is safety critical, the whole computation is done in rational arithmetic to avoid misclassification by rounding errors.
Therefore several heuristic methods were developed to find interpolants with as few linear constraints as possible. Scholl et al. [48] were able to outperform several existing tools in the computation of quantifier eliminations, by only removing redundant constraints, i.e. constraints that are unnecessary to describe the current non-convex polyhedra. After these results Scholl et al. [49] developed another algorithm that computes *simple interpolants* by combining theory lemmata of a resolution proof, to minimize the number of linear con-

straints used in such an interpolant. Another algorithm was proposed by Albarghouthi and McMillan [2], who compute an interpolant from scratch by iteratively exploring the non-convex polyhedra and separating pairs of convex regions. This, combined with a clever combination of merging and splitting these pairs together, ends up in a method which is most effective when the resulting interpolants consist of few linear constraints and no discrete state space. Due to the fact that our state space has a large number of discrete variables, this method has to be adapted to be applicable on our problems.

1.1 CONTRIBUTIONS AND OBJECTIVES

The main contribution of this part of this thesis is the development of a heuristic for improving the number of linear constraints for an interpolant of two non-convex polyhedra.
Additionally we present an algorithm that computes lower bounds for the number of linear constraints used in these interpolants in very specific situations. Those lower bounds can be used to verify global optimality of certain interpolants and therefore we can distinguish between cases where there is potential room for optimization and those which are not improvable.
Further we design an algorithm that reduces the number of bits that the worst coefficient of a linear constraint is using in its description, while again limiting the overapproximation by another state set similar to the limitation where we reduce the linear constraints.
Parts of the main contribution, the algorithm for improving interpolants, were published in [3]. Furthermore the CEGAR-approach, i.e. the algorithm in which this procedure is used, was published in [4].

1.2 CONTEXT

This part of the thesis was mainly supported by the German Research Foundation (DFG) in the context of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, [7]). It was part of the subproject "First Order Model Checking" within the subproject H3, which focused on the automatic abstraction for hybrid controllers.

1.3 OUTLINE

The remainder of the first part of this thesis is structured as follows. In Chapter 2 we give general preliminaries. We focus on linear optimization, but we also give an overview of well-known techniques used to solve satisfiability modulo theory formulas, and a basic insight of the related model-checking algorithm.

In the section for linear programming we will state certain milestones leading to the well-known simplex method, which is the commonly used technique in state-of-the-art Linear Program (LP) solvers. Besides the fact that we will use this method for solving every LP in this thesis it is also a fundamental basis for the concept of column generation, which we will introduce in the second part of the thesis. Along with this method we will introduce the basic concept of Farkas' Lemma, which is the basis for the LP formulation in the core of our algorithm.
The section about satisfiability modulo theory gives an overview of the syntax and semantics, as well as basic insights of techniques used in SMT solvers. While many of this is true for general theories we focus only on the theory of real and rational linear arithmetic.
In the section about the model-checking algorithm, we will roughly sketch the algorithm, but we focus on the representation of state sets and the situations in which circumstances an abstraction or a refinement must be computed.
Chapter 3 gives a formal definition of the problem and a sketch why this problem is NP-hard.
In Chapter 4 we will briefly introduce the abovementioned related algorithms that compute Craig interpolants to find a compact form for state sets. While the methods in Section 4.1 and 4.3 where also explicitly developed for this model-checking algorithm, the method described in Section 4.2 is an approach published for verification purposes, but not explicitly for this context.
Chapter 5 presents our developed algorithm for solving the minimization interpolant problem heuristically. Instead of computing the actual description of our new state set directly, we minimize a set of linear constraints while preserving a property that implies that a representation for the state set can be build by only using this set of linear constraints. With that in mind we minimize the set by successively replacing two linear constraints by one new linear constraint. We also include several strategies to optimize the procedure and to generalize it for open or partially open state sets.
In Chapter 6 we describe a procedure to compute lower bounds for the number of constraints used in an interpolant. Especially for a situation which often occurs due to a refinement step in the CEGAR-algorithm.
As a last application we propose an algorithm in Chapter 7 that reduces the maximum number of bits used in a linear constraint present in an interpolant.
Chapter 8 presents the experiments we made to evaluate our optimization strategies, and to compare our algorithm with related algorithms. Additionally we evaluate the algorithm for the computation for lower bounds, and the description test.
In Chapter 9 we summarize the results of our algorithm and discuss

the differences between our experimental results and the results of our work group and why both experiments can differ in results.

2

PRELIMINARIES

In this chapter we want to introduce notions that we use throughout the thesis, followed by an introduction of linear programming. After that we shortly introduce techniques and theories that we only use in the part of model-checking, i.e. Satisfiability Modulo Theory and Fourier–Motzkin elimination. This content is not original and is mentioned for the completeness of the work and a better understanding for the decisions made in our algorithms.

2.1 NOTATIONS

The symbols $\mathbb{R}, \mathbb{Q}$ and $\mathbb{N}$ denote the real, rational and natural numbers. $\mathbb{B} = \{0, 1\}$ represents the Boolean values *true* and *false*, also stated as $\top$ and $\bot$.
A matrix of dimensions $m \times n$ is an array of real numbers a_{ij}:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

For an arbitrary matrix A with fixed dimensions $m \times n$ we denote $A \in \mathbb{R}^{m \times n}$. The (i, j)-th entry of a matrix A is denoted by a_{ij} or $[A]_{ij}$.
A *row vector* is a matrix where $m = 1$, a *column vector* or simply *vector* where $n = 1$. For an n-dimensional vector x, denoted by $x \in \mathbb{R}^n$, we use x_i to denote its i-th component. The n-dimensional vector with all components equal to zero we denote by 0^n, or simply by 0. For the i-th *unit vector*, i.e. the vector with all components equal to zero except that the i-th component is equal to one, is denoted by e_i.
The *transpose* A^T of an $m \times n$ matrix A is the $n \times m$ matrix

$$A^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}.$$

It therefore holds that $\left[A^T\right]_{ij} = [A]_{ji}$.
The inner product of two vectors $x, y \in \mathbb{R}^n$ is defined by

$$x^T y = y^T x = \sum_{i=1}^{n} x_i y_i.$$

For a matrix A we denote its j-th column by $A_j = \left(a_{1j}, a_{2j}, \ldots, a_{mj}\right)^T$. Similar we will denote its i-th row by $a_i = (a_{i1}, a_{i2}, \ldots, a_{in})$. Therefore A can be written as

$$A = \begin{pmatrix} | & | & & | \\ A_1 & A_2 & \ldots & A_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{pmatrix} .$$

For two matrices A, B with dimensions $m \times n$ and $n \times k$, their product AB results in a matrix of dimensions $m \times k$, with the following entries

$$[AB]_{ij} = \sum_{l=1}^{n} [A]_{il} [B]_{lj} = a_i^T B_j .$$

If the number of columns and rows are equal the matrix is called *square*. The identity matrix in $n \times n$, i.e. the matrix whose entries are all equal to zero except the ones on its diagonal are one, is denoted by I^n. Let A be an n-dimensional square matrix. If there exists a square matrix B of the same dimension, satisfying $AB = BA = I^n$, the matrices are called *invertible*, and B is called the inverse of A. The inverse matrix is unique and denoted by A^{-1}.
For a vector x and an (in-)equality sign $\sim$ the notation $x \sim 0$ means that for all components $x_i \sim 0$ the in-/equality holds. Similar for two vectors $x, b \in \mathbb{R}^n$ and an in-/equality sign $\sim$ the notation $x \sim b$ means that for all components $x_i \sim b_i$ the in-/equality holds.

We describe a linear constraint l in $\mathbb{R}^m$ by a formula $d^T x \leqslant d_0$ with real coefficients $d_0, d_1, \ldots, d_m \in \mathbb{R}^{m+1}$ over real variables $x_1, \ldots, x_m$. Let P be a set of m linear independent points in $\mathbb{R}^m$, then P induces two linear constraints l_P^+ and l_P^-, such that all $p \in P$ fulfill the constraint with equality. Sometimes we just want to refer to any of these two linear constraints, then we will denote it as l_P. Therefore we can interpret any linear constraint l in $\mathbb{R}^m$ as a function

$$l : \mathbb{R}^m \to \mathbb{B} \quad p \mapsto \begin{cases} 0 & \text{if } d^T p > d_0 \\ 1 & \text{if } d^T p \leqslant d_0 \end{cases} .$$

Additionally a linear constraint l defines different subsets of the real space $\mathbb{R}^m$. For each symbol $\sim \in \{<, \leqslant, =, \geqslant, >\}$ we define the subset

$$l_\sim = \{x \in \mathbb{R}^m | d^T x \sim d_0\} .$$

If we speak about the hyperplane l we usually mean $l_=$. We write $Dx \leqslant d_0$ for a conjunction of u linear constraints over the variables

$(x_1, \dots, x_m)^T = x$ with $D \in \mathbb{R}^{u \times m}$ and $d_0 \in \mathbb{R}^u$.
Furthermore a point of a state set is an element of $\mathbb{B}^n \times \mathbb{R}^m$, therefore we denote the first n components of p by $p_{\mathbb{B}}$, and the last m by $p_{\mathbb{R}}$. The linear constraint and the function described above are always defined only on the part of the point $p_{\mathbb{R}}$. Since the linear constraints are only based on the real part we will sometimes write p instead of $p_{\mathbb{R}}$.

2.2 LINEAR PROGRAMMING

An LP is given by

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & A^{(1)} x \leqslant b^{(1)} \\ & A^{(2)} x = b^{(2)} \\ & A^{(3)} x \geqslant b^{(3)} \,. \end{aligned} \tag{1}$$

The linear function $c^T x$ is called *objective function*, while all equalities and inequalities used in the description are usually called *constraints*. A variable x that fulfills the constraint $x \geqslant 0$ is denoted as *positive*; and *negative* when the constraint $x \leqslant 0$ holds. If both are not valid the variable is called *free*.
Each x that satisfies every linear constraint and the variable bounds is called a *feasible solution*. The set of all these feasible solutions is called the *feasible set*. If a feasible solution x^* minimizes the objective function, i.e. it holds that $c^T x^* \leqslant c^T x$ for every feasible solution x, it is called an *optimal solution*. The corresponding objective value is called an *optimal value*. If there exists no lower bound for the objective value the problem is called *unbounded*. In case that the feasible set is empty it is called *infeasible*.
The constraints of an LP define a polyhedron in $\mathbb{R}^n$, therefore we will quickly review some aspects of polyhedra.

2.2.1 *Polyhedra*

A *Polyhedron* $P \subseteq \mathbb{R}^n$ is a set that can be described by the intersection of finitely many linear inequalities or equalities, i.e.

$$P = \left\{ x \in \mathbb{R}^n \;\middle|\; \begin{array}{rcl} A^{(1)} x & \leqslant & b^{(1)} \\ A^{(2)} x & = & b^{(2)} \\ A^{(3)} x & \geqslant & b^{(3)} \end{array} \right\} .$$

A polyhedron is *bounded* if there exists a constant W, such that $P \subseteq [-W, W]^n$. A bounded polyhedron is also called *polytope*.
For a fixed point we say a constraint of the polyhedron is *active* or

binding, if the constraint is fulfilled with equality at this point. A vector $x^* \in \mathbb{R}^n$ is a *basic solution* if all equality constraints are active, and out of every constraints that are active there exist n that are linear independent. Furthermore if a basic solution is feasible, we say that it is a *basic feasible solution*. A vector $x \in P$ is a *vertex* if there exists some objective function defined by a vector $c \in \mathbb{R}^n$, where x is the only optimal solution in P when minimizing over c. We call a vector $x \in P$ an *extreme point* of P, if there exist no two points $y, z \in P$ with $x \neq y \neq z$ and a scalar factor $\lambda \in [0, 1]$, such that $x = \lambda y + (1 - \lambda) z$. The following theorem states that all three definitions are equivalent if P is non-empty.

Theorem 1.
Let P be a non-empty polyhedron and let $x^* \in P$. Then, the following are equivalent:

a) x^* *is a vertex;*

b) x^* *is a extreme point;*

c) x^* *is a basic feasible solution.*

There are two different specific representations of a polyhedron used in linear programming, the canonical form and the standard form. We will show that both representations can be achieved for any polyhedron and therefore are no restrictions.
The *canonical form* uses only greater-than-or-equal-to constraints. Therefore we reformulate every equality constraint $(a, b) \in (A_2, b_2)$, by two inequality constraints $a^T x \leqslant b$ and $a^T x \geqslant b$. Further, for every $(a, b) \in (A_1, b_1)$, we will add $-a^T x \geqslant -b$ to the constraints. Finally the only constraints left are $A^{(3)} x \geqslant b^{(3)}$. Notice that in each step we will modify the matrices $A^{(i)}$ and the right-hand-side vectors $b^{(i)}$, i.e. we will delete the constraint that we reformulate and inserting the new constraint to the appropriate matrix and right-hand-side vector.
The *standard form* uses only equality constraints and positive variables. First we eliminate all free variables, by introducing two new positive variables $x^+, x^- \geqslant 0$ for each free variable x and substitute every appearance of x by $x^+ - x^-$. Second we eliminate all inequality constraints, by introducing so called *slack and surplus variables*. For every $(a, b) \in (A_1, b_1)$ we introduce a positive slack variable s and replace the constraint by the equality constraint $a^T x + s = b$. Similar for every $(a, b) \in (A_3, b_3)$ we introduce a surplus variable s and replace the constraint by $a^T x - s = b$.
Both reformulations will still describe the same sets in $\mathbb{R}^n$ and therefore the optimization problem will have the same optimal value and solution in every representation.

2.2.2 *Simplex Algorithm*

There are multiple ways to solve linear programs, we focus on the most commonly used in practical applications, the simplex algorithm first described by Dantzig in 1948 [20–22]. It is worth mentioning that the worst case runtime of this method is exponential and that there exist weakly polynomial algorithms that solve linear programs, i.e. the ellipsoid method [36] and the interior point method [32]. While the ellipsoid method is only of theoretical interest and no implementation can compete in usual applications against the simplex method, there exist applications in which the interior point method is comparable or even faster than the simplex algorithm.
The rough idea of the simplex algorithm is to go from one vertex to another, always checking if there is an adjacent vertex with a better objective value, or in fact if there is a direction in which the objective value improves. For the rest of this section we consider the LP in standard form,

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geqslant 0\,, \end{aligned} \tag{2}$$

where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and $c, x \in \mathbb{R}^n$. We can assume that A has a full row rank, otherwise there exists another matrix A' with full rank, which describes the same polyhedron as A. A set of indices $B = \{B_1, \ldots, B_m\}$ and the set $N = \{1, \ldots, n\} \setminus B$, the induced matrix $A_B = \left(A^{B_1}, \ldots, A^{B_m}\right)$ is called *basis*, if the columns are linearly independent. Often the set of indices B is called *basis*, too. An initial basis is often known for a specific LP and most solvers will gladly take such information into account. Otherwise there is always the possibility to solve an *auxiliary* LP to get such a basic feasible solution. We assume, without loss of generality that $b \geqslant 0$, we can always switch the sign of b_i by multiplying its equality constraint by -1. Further we introduce positive *artificial variables* $y \in \mathbb{R}^m$ for every constraint, and add y_i on the left-hand-side of the i-th constraint. The auxiliary problem is finally

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} y_i \\ \text{s.t.} \quad & Ax + y = b \\ & x \geqslant 0 \\ & y \geqslant 0\,. \end{aligned} \tag{3}$$

This is a problem in standard form, and by setting $x = 0$ and $y = b$, we always have a feasible basic solution at hand and therefore can use the simplex algorithm to solve it. Since we penalize the use of the variables in y, any feasible solution that solves the initial problem (2)

has the objective value of 0. If the problem terminates with an optimal value greater than zero, we can conclude that there exists no feasible solution to the original problem.
So we can consider that we have a basic feasible solution x, i.e. all nonbasic variables are zero, while $x_B = (x_{B_1}, \dots, x_{B_m})$ is given by

$$x_B = A_B^{-1} b \ .$$

Now we want to find an adjacent vertex to x by the construction of a *feasible direction*.

Definition 2.
Let x be an element of a polyhedron P. A vector $d \in \mathbb{R}^n$ is called feasible direction *at x, if there exists a positive scalar τ for which $x + \tau d \in P$.*

Therefore we select a nonbasic variable x_j and increase it to a positive value τ, while all other nonbasic variables remain at zero. In fact we set the components of a vector d, such that $d_j = 1$ and all components for other nonbasic variables to zero. Then the vector of the basic variables x_B changes into $x_B + \tau d_B$, where $d_B = (d_{B_1}, \dots, d_{B_m})$. Since we want to find a new basic feasible solution, and therefore a feasible solution, the new point must satisfy the constraints, i.e. $A(x + \tau d) = b$. This can be transformed to $Ad = 0$, since x is a feasible solution and $\tau > 0$. We can transform this even further by recalling the construction of d

$$\begin{aligned} 0 &= Ad \\ &= \sum_{i=1}^{n} A_i d_i \\ &= \sum_{i=1}^{m} A_{B_i} d_{B_i} + A_j \\ &= A_B d_B + A_j \ . \end{aligned}$$

With the fact that B is invertible, we obtain

$$d_B = -B^{-1} A_j \ .$$

Thus the equality constraints hold for this computed direction. The non-negativity constraints for all variables are initially valid. Further all nonbasic variables are increased and are therefore valid. Thus we only have to determine τ such that the first basic variable hits zero.
In the degenerate case it is possible that d is not a feasible direction, and therefore τ is determined as zero.
To determine if it is valuable for the objective function to choose the j-th nonbasic variable, we introduce the so called *reduced cost* of the j-th variable.

Definition 3.
Let x be a basic solution, let A_B be the associated basis matrix, and let c_B be

the vector of costs of the basic variables. For each j*, we define the* reduced cost $\bar{c}_j$ *of the variable* x_j *according to the formula*

$$\bar{c}_j = c_j - c_B^T A_B^{-1} A_j \, .$$

Consider a basic variable x_{B_i}, we can determine the reduced costs by the fact that $A_B^{-1} A_{B_i}$ is the i-th unit vector e_i. Therefore the reduced costs are

$$\bar{c}_{B_i} = c_{B_i} - c_B^T A_B^{-1} A_{B_i} = c_{B_i} - c_B^T e_i = c_{B_i} - c_{B_i} = 0 \, .$$

Thus the reduced costs for every basic variable is zero. The next theorem states the connection between the vector of all reduced costs and the optimality of the solution.

Theorem 4.
Consider a basic feasible solution x *associated with a basis* B*, and let* $\bar{c}$ *be the corresponding vector of reduced costs.*

a) If $\bar{c} \geqslant 0$*, then* x *is optimal.*

b) If x *is optimal and not degenerate, then* $\bar{c} \geqslant 0$*.*

So if we choose a nonbasic variable j we can determine a specific direction d associated with j and the reduced costs $\bar{c}_j$. If $\bar{c}_j < 0$ and therefore indicates that we improve the objective function in the direction d, we now have to determine how far we move in this direction. Since the costs improve in the direction, we want to maximize the way, while staying feasible. The new point is computed by $x + \tau d$, with $\tau \geqslant 0$, therefore we have to compute,

$$\tau^* = \max\{\tau \geqslant 0 \mid x + \tau d \in P\} \, .$$

There are two cases we have to distinguish:

a) If $d \geqslant 0$, then $x + \tau d \geqslant 0$ for all $\tau \geqslant 0$, and the problem is therefore unbounded.

b) There exists an i such that $d_i < 0$, and the constraint $x_i + \tau d_i \geqslant 0$ gives an upper bound for $\tau \leqslant -x_i/d_i$. For each negative component i we therefore obtain an upper bound $-x_i/d_i$, and τ^* is the lowest of all of these bounds, i.e.

$$\tau^* = \min_{\{i \mid d_i < 0\}} (-x_i/d_i) = \min_{\{i=1,\ldots,m \mid d_{B_i} < 0\}} (-x_{B_i}/d_{B_i}) \, .$$

Since we set the component d_i to zero for all nonbasic variables except the entering variable j, we only have to consider basis variables B_i. Note that if B is not degenerate, all $x_{B_i} > 0$, and therefore $\tau^* > 0$.

Let us assume that all basic feasible solution are not degenerate. Then we are able to give a typical iteration of the simplex method, usually denoted as *pivot* that will improve the objective function or terminate.

a) The iteration starts with a basic feasible solution x, with an associated basis B.

b) Compute the reduced costs $\bar{c}_j$ for all nonbasic variables j. If one is negative we can improve the objective function, otherwise Theorem 4 states that x is optimal.

c) Compute the direction $d_B = B^{-1}A_j$ for a variable j with negative reduced costs. If there exists no negative component in d_B we terminate as the problem is unbounded.

d) If a component in d_B is negative, we compute

$$\tau^* = \min_{\{i=1,\dots,m \mid d_{B_i}<0\}} (-x_{B_i}/d_{B_i}) \ .$$

e) Let l be the index for that $\tau^* = -x_{B_l}/d_{B_l}$ holds, which means B_l is leaving the basis. Then the new basic feasible solution x' is computed by $x'_j = \tau^*$ and $x'_{B_i} = x_{B_i} + \tau^* d_i$ for $i \neq l$.

Notice that in step (c) we have the freedom to choose any of the variables that have negative reduced costs. In Figure 1 we show a possible route through some vertices of a polyhedron by a simplex algorithm.

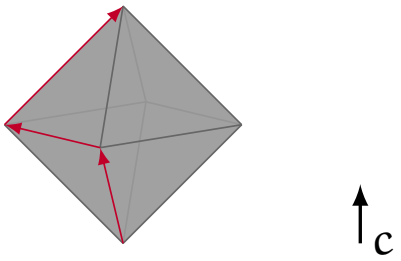


Figure 1: Example route of a simplex algorithm

In case there exist basic feasible solutions that are degenerate we have to specify how to choose the incoming variable in the basis. There are specific rules, so called *pivoting rules*, which prevent the algorithm from an indefinite loop while changing the basis but not the basic feasible solution x. This undesirable effect is called *cycling*. Perhaps the simplest out of all pivoting rules is to take the *smallest subscript*, i.e. choose the smallest j for which $\bar{c}_j$ is negative.

2.2.3 *Duality of Linear Programs*

For several theoretical reasons we want to introduce the dual theory of linear programs. Let an LP in standard form be given, i.e.

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geqslant 0 \ . \end{aligned} \tag{4}$$

Table 1: Relation between variables and constraints in primal and dual LPs

primal	minimize	maximize	dual
constraints	$\geqslant b$	$\geqslant 0$	variables
	$\leqslant b$	$\leqslant 0$	
	$= b$	free	
variables	$\geqslant 0$	$\leqslant c$	constraints
	$\leqslant 0$	$\geqslant c$	
	free	$= c$	

We call this the *primal problem* (P). Then the following problem

$$\begin{aligned} \max \quad & b^T y \\ \text{s.t.} \quad & A^T y \leqslant c \end{aligned} \tag{5}$$

is called its *dual problem* (D). In general we can note that for every constraint in (P), there exists a variable in (D), and additionally for every variable in (P) there exists a constraint in (D). As another fact the sign of the constraint and the sign of the variable are related. Table 1 states how the different kinds are related. As in the general transformation of LPs we can always transform a maximization problem into a minimization problem by taking $-c$ instead of c as the objective function. With this in mind, computing the dual of (D) leads to Lemma 5

Lemma 5.
Let (P) *be an* LP *and* (D) *its dual* LP. *Then the dual* LP *of* (D) *is equivalent to* (P).

We want to state some major theorems known for the duality theory of LPs.

Theorem 6 (Weak duality)**.**
If x *is a feasible solution for the primal problem* (P) *and* y *a feasible solution for the dual problem* (D), *then*

$$b^T y \leqslant c^T x \,.$$

From weak duality several simple corollaries follow. For instance that if one of the problems is unbounded the other has to be infeasible. Another corollary states that if there are feasible solutions for both problems with the same objective value, those solutions have to be optimal for their problem.

Theorem 7 (Strong duality)**.**
If an LP *has an optimal solution, so does its dual, and the optimal costs of both are equal.*

Recall that for every LP there are three possible outcomes, where exactly one will occur:

a) There is a finite optimal solution.

b) The problem is *unbounded*, i.e. the optimal cost is $-\infty$ (minimization), or ∞ (maximization).

c) The problem is *infeasible*, i.e. no feasible solution exists.

This leads to nine possible outcomes for the primal and the dual. In fact these outcomes are not all possible. We already excluded four combinations by a corollary of weak duality. Finally with the help of the theorem of strong duality we can limit the number of possible outcomes to four situations:

1. Both the primal and the dual problem have a finite optimum.
2. The primal problem is unbounded, and the dual problem is infeasible.
3. The dual problem is unbounded, and the primal problem is infeasible.
4. Both problems are infeasible.

The theorems of weak and strong duality were statements over the objective value of feasible or optimal solutions. The next theorem provides a relation between the primal and dual solutions.

Theorem 8 (Complementary slackness)**.**
Let x and y be feasible solutions of (P) and (D), respectively. They are optimal solutions for the two respective problems if and only if:

$$y_i(a_i^T x - b_i) = 0, \qquad \forall i,$$
$$(c_j - y^T A_j)x_j = 0, \qquad \forall j.$$

2.2.4 *Farkas' Lemma*

Suppose that we want to determine if a given system of linear inequalities is infeasible. We will use the duality theory to show that the infeasibility of a given system is equivalent to the feasibility of another system of inequalities. Consider that the first system is given by a system in standard form, i.e. $Ax = b$ and $x \geqslant 0$. Farkas' lemma states that either there exists such a solution, or there exists a certificate of infeasibility, i.e. a solution y such that $A^T y \geqslant 0$ and $y^T b < 0$.

Theorem 9 (Farkas' Lemma)**.**
Let A be a matrix of dimensions $m \times n$ and let $b \in \mathbb{R}^m$. Then, exactly one of the following two statements holds:

a) There exists some $x \geqslant 0$ *such that* $Ax = b$.

b) There exists some vector y *such that* $A^T y \geqslant 0$ *and* $y^T b < 0$.

Proof. We have to prove two directions, if there exists $x \geqslant 0$ satisfying $Ax = b$, and if $A^T y \geqslant 0$, then we can conclude that

$$y^T b = y^T A x \geqslant 0 \, .$$

This concludes that if (a) holds, (b) is not satisfiable. As we stated above we want to use the duality theory, therefore we need LPs to use the theorems. We will formulate the primal problem in standard form, with the objective function $c = 0$, and formulate it as a maximization problem. Then we can state the primal and the dual problem,

$$\begin{array}{lll} \max & 0^T x & \qquad \min \quad b^T y \\ \text{s.t.} & Ax = b & \qquad \text{s.t.} \quad A^T y \geqslant 0 \, . \\ & x \geqslant 0 \, , & \end{array}$$

Now let us assume that the primal problem is infeasible, i.e. there exists no $x \geqslant 0$ such that $Ax = b$. The infeasibility of the primal implies that the dual problem is either unbounded or itself infeasible. Since $y = 0$ implies that $A^T y = 0 \geqslant 0$, this is a feasible solution for the dual problem. Though the objective costs are $-\infty$, and so there must exist a feasible solution y such that $y^T b < 0$. □

We will later use a modification of Farkas' Lemma to find indistinguishable pairs in our algorithm. There our primal problem is given in the form where only lesser-then-or-equal constraints are used, i.e. $Ax \leqslant b$. And we already know that we describe two feasible polyhedra $\mathcal{A}$ and $\mathcal{B}$, which are disjoint.
For this context we will use a theorem that is a variant of Farkas' Lemma. This is one amongst other theorems that are often called *theorems of the alternative*.

Theorem 10.
Suppose that the system of linear inequalities $\mathcal{A}x \leqslant \alpha$ *has at least one solution, and let* d_0 *be some scalar. Then, the following are equivalent:*

(a) Every feasible solution to the system $\mathcal{A}x \leqslant \alpha$ *satisfies* $d^T x \leqslant d_0$.

(b) There exists some $y \geqslant 0$ *such that* $\mathcal{A}^T y = d$ *and* $y^T \alpha \leqslant d_0$.

Proof. Consider the pair of LPs

$$\begin{array}{lll} \max & d^T x & \qquad \min \quad \alpha^T y \\ \text{s.t.} & \mathcal{A}x \leqslant \alpha \, , & \qquad \text{s.t.} \quad \mathcal{A}^T y = d \\ & & \qquad\qquad y \leqslant 0 \, . \end{array}$$

and note that they are dual to each other. Since the first one is feasible by assumption, it can only be bounded or unbounded. If (a) holds, then it is bounded by d_0 and therefore there exists an optimal solution. Strong duality gives the fact that the optimal solution of the dual problem must be lower than d_0, too. Therefore there exists a solution of the dual that satisfies (b). If (b) holds, then weak duality states that the left LP is bounded by the objective value of the dual so (a) holds. □

We want to modify this theorem such that we combine two systems of linear inequalities $\mathcal{A}x \leqslant \alpha$ and $\mathcal{B}x \leqslant \beta$, which both have at least one feasible solution. But they are not simultaneously satisfiable.

Theorem 11.
Suppose the two systems of linear inequalities $\mathcal{A}x \leqslant \alpha$ and $\mathcal{B}x \leqslant \beta$ have both at least one feasible solution. Then, the following are equivalent:

a) Every feasible solution of the system $\mathcal{A}x \leqslant \alpha$ satisfies $d^T x \leqslant d_0$ and every feasible solution of the system $\mathcal{B}x \leqslant \beta$ satisfies $d^T x \geqslant d_0$.

b) There exists some $y = (y_a, y_b) \geqslant 0$ such that $A^T y_a = d$, $B^T y_b = -d$, $y_a^T \alpha \leqslant d_0$ and $y_b^T \beta \leqslant -d_0$.

Proof. Assume (a) is true, then Theorem 10 states for the system of linear inequalities $\mathcal{A}x \leqslant \alpha$ that there exists $y_a \geqslant 0$ such that $\mathcal{A}^T y_a = d$ and $y_a^T \alpha \leqslant d_0$, which is part of the statement of (b). When we use the other system of linear inequalities $\mathcal{B}x \leqslant \beta$, and modify the statement $d^T x \geqslant d_0$ to $-d^T x \leqslant -d_0$, Theorem 10 states that there exists $y_b \geqslant 0$ such that $\mathcal{B}^T y_b = -d$ and $y_b^T \beta \leqslant -d_0$, which gives us the missing part of the statement of (b).
Consider (b) is true, then again Theorem 10 gives the statement of (a) for both linear systems separately. □

Note that we can solve (b) by a linear program. Additionally d, d_0 can be set as variables without creating a non-linear constraint. Since condition (b) is only a statement of feasibility, there is no objective function used so far. This leads to more freedom in terms of modeling other goals. In Section 5.1.3 we describe an LP that solves a similar problem based on the idea of this theorem.

2.3 SATISFIABILITY MODULO THEORY

Satisfiability modulo theory (SMT) is an extension of the common known SAT problem. In SAT problems there is a given formula ϕ in propositional logic, i.e. a formula built by Boolean variables, where the operators are conjunction $(\wedge)$, disjunction $(\vee)$, negation $(\neg)$, and parentheses. A formula ϕ is satisfiable, if there exists a model m, i.e. an assignment of the variables, such that the formula evaluated in m is true.

In SMT problems, some of the Boolean variables have a deeper meaning, i.e. must fulfill some background theory, e.g. real numbers, arithmetic, or linear arithmetic. The basics of SMT can be backtracked to the 1970s and 1980s, where decision procedures where used in formal methods, see [11, 45, 51]. The modern SMT solvers combine fast SAT solvers with theory solvers, which typically only solve conjunctions of literals. To formally introduce SMT formulas we first give basic concepts and notations about first-order logic, then we specify the setting by introducing only the theory of linear arithmetic.

2.3.1 *Syntax*

A *signature* Σ contains *predicate* and *function* symbols, and every symbol has a specific *arity*. Further we partition any signature Σ into Σ^F and Σ^P, denoting that they are functions or respectively predicate symbols. A 0-arity symbol in Σ^F is called a *constant* symbol. The 0-arity symbols in Σ^P are called *propositional* symbols or *Boolean variables*. A *term* t is a first-order term built from the function symbols of Σ. As it is common, we call a formula *atomic* (or *atom*), if it has one of the following forms, a 0-arity symbol in Σ^P, a propositional n-arity $p(t_1, \ldots, t_n)$ with terms $t_1, \ldots, t_n$, an equality $t_1 = t_2$, false $\bot$, or true $\top$. A *literal* is an atom or the negation of an atom. The *complement* of a literal l is written as $\neg l$, such that if l is an atom it is the negation, and if l is already the negation of an atom, it is the atom itself. The disjunction of literals $l_1 \vee \cdots \vee l_n$ is called a *clause* c, we then denote with $\neg c$ the set $\{\neg l_1, \ldots, \neg l_n\}$. Correspondingly for a set of literals $\mu = \{l_1, \ldots, l_n\}$, we denote by $\neg \mu$ the clause $\neg l_1 \vee \cdots \vee \neg l_n$. A formula is in Conjunctive Normal Form (CNF), or called a CNF-formula, if it is a conjunction of clauses, e.g. $c_1 \wedge \cdots \wedge c_n$.

2.3.2 *Semantics*

Formulas are mapped to the set of truth values $\{\mathtt{true}, \mathtt{false}\}$, with the concept of *models*. A model $\mathcal{A}$ for a signature Σ is a pair $(A, (\cdot)^{\mathcal{A}})$, where A is a non-empty set and $(\cdot)^{\mathcal{A}}$ a mapping. This mapping assigns every constant to an element in A, every symbol in Σ^F with n-arity ($n > 0$) to a total function that maps A^n to A. Additionally this mapping assigns each propositional to $\{\mathtt{true}, \mathtt{false}\}$, and every symbol in Σ^P with n-arity ($n > 0$) to a total function that maps A^n to $\{\mathtt{true}, \mathtt{false}\}$. Generally the mapping $(\cdot)^{\mathcal{A}}$ is extended to map terms to an element of A and formulas to an truth value $\{\mathtt{true}, \mathtt{false}\}$. This extension is called *interpretation* of the terms and formulas. A model $\mathcal{A}$ satisfies a formula ϕ if and only if, $\phi^{\mathcal{A}}$, the interpretation of the formula, is true.
Now in the context of SMT we are no longer interested in any model that satisfies a formula, but in a model belonging to a given theory $\mathcal{T}$,

which constraints the interpretation of the symbols of Σ. In this context say a ground formula ϕ is satisfiable in $\mathcal{T}$, if and only if there is an element of the set $\mathcal{T}$ that satisfies ϕ. Similar to this, a set of ground formulas Γ ($\mathcal{T}$)-*entails* a ground formula ϕ, if and only if every model of $\mathcal{T}$ that satisfies every formula in Γ satisfies ϕ as well, written $\Gamma \models_{\mathcal{T}} \phi$. Such sets Γ are called $\mathcal{T}$-*consistent* if and only if $\Gamma \not\models_{\mathcal{T}} \bot$, and a function ϕ is $\mathcal{T}$-*valid* if and only if $\emptyset \models \phi$. We therefore call a clause C a *theory lemma* if C is $\mathcal{T}$-valid.
For a set of literals C and a formula F, we denote by $C \setminus F$ the set containing the literals in C by removing all atoms occurring in F, and by $C \downarrow F$ the set of literals created from C by removing all atoms *not* occurring in F.

2.3.3 *Real Linear Arithmetic*

Since the theory of linear arithmetic $\mathcal{LA}(\mathbb{Q})$ is already a combination of theories, i.e. rational numbers and linear arithmetic, we would have to formally introduce both and then talk about the combination of theories. For the sake of simplicity we will skip this part and leave it to the reader to get additional information if needed, details are shown in [6]. For rational linear arithmetic we now have two disjoint sets of variables C and B, where C contains continuous variables $c_1, \ldots, c_m \in \mathbb{Q}$, and B contains Boolean variables $b_1, \ldots, b_n \in \{0, 1\}$. Then we want to reason about formulas over $B \cup C$, but with the restriction that every term over C has to be of the form of a linear term, i.e. there are rational constants $\alpha_0, \ldots, \alpha_m$ such that the term is of the kind $\alpha_0 + \sum_{i=1}^{m} \alpha_i c_i$. The predicates are then based on linear constraints, i.e. for a term t they have the form $t \sim 0$ with $\sim \in \{=, <, \leqslant >, \geqslant\}$. As a remark, if we only allowed $\sim$ to be $\leqslant$ it would not reduce the expressive power of the theory, since every other symbol can be expressed by $\leqslant$ with the negation and combination of Boolean formulas. As an example an equation is simply a conjunction of two linear constraints, and the strict inequality is just the negation of non-strict equation with negated factors. An (quantifier-free) SMT formula is therefore a formula over the set of all Boolean combinations of the Boolean variables B and linear constraints over C.
A basic example of an SMT formula in real linear arithmetic is

$$((c_1 + c_2 \geqslant 0) \vee b_1) \wedge (((c_1 < 0) \wedge (c_2 < 0)) \vee b_2) \wedge \neg (b_1 \wedge b_2)\,. \quad (6)$$

2.3.4 *Typical Solving Strategies for* SMT*-Formulas*

In this subsection we want to introduce some basic strategies that are used to solve SMT formulas. Additionally we want to point out which tools are usually available in common SMT solvers, e.g. incremental solving and resolution proofs. A common way is to represent every linear constraint, or in general every atom, as a new Boolean variable.

Then solve the formula by an SAT solver, since the formula now only consists of Boolean variables. Convert the model that was found for the SAT problem back to the theory and check this for consistency. If it is consistent in the theory, the solver found a solution. If it is not consistent, the theory solver will derive a cause, i.e. a subset of atoms that cannot be fulfilled at the same time. We will call this subset a $\mathcal{T}$-*conflict*. Then the solver will again substitute this $\mathcal{T}$-*conflict* to the Boolean variables and will add the negation as a conjunction to the formula. Then the SAT solver will again try to solve this. The procedure stops when the SAT solver cannot come up with a model, i.e. will state the formula is unsatisfiable, or the theory solver states that the model is consistent in the given theory, where it will state that the formula is satisfiable.

Example 1.
Assume an SMT *solver wants to check if the formula in (6) is satisfiable. Then it would substitute every atom, i.e.*

$$q_1 \mapsto c_1 + c_2 \geqslant 0 \qquad q_2 \mapsto c_1 < 0 \qquad q_3 \mapsto c_2 < 0\,.$$

The SAT *solver will now solve the formula*

$$(q_1 \vee b_1) \wedge ((q_2 \wedge q_3) \vee b_2) \wedge \neg (b_1 \wedge b_2)\,, \tag{7}$$

and find a Boolean assignment, e.g. $q_1, q_2, q_3 = \top$ *and* $b_1, b_2 = \bot$. *Now the theory solver has to decide whether this Boolean assignment (i.e. the substitution back to the linear constraints) is consistent with the theory or not, i.e. if there exists a solution for* c_1, c_2, *such that* $q_1, \ldots, q_3 = \top$. *Obviously in this situation this is not possible and the solver returns a conflict. In our case the conflict is* $\{(c_1 + c_2 \geqslant 0), (c_1 < 0), (c_2 < 0)\}$. *Then the conflict is added to the* SAT *formula in (7), which leads to*

$$(q_1 \vee b_0) \wedge ((q_2 \wedge q_3) \vee b_1) \wedge \neg (b_0 \wedge b_1) \wedge (\neg q_1 \vee \neg q_2 \vee \neg q_3)$$

The solver will return a new model, e.g. $q_1, b_2 = \top$ *and* $q_2, q_3, b_1 = \bot$. *And since this is consistent with the theory, the theory solver will return a suitable assignment, e.g.* $c_1 = c_2 = 1$.

2.3.4.1 *Incremental Solving*

Most common SMT solvers will process the Boolean model incremental, and propagate if a partial model is consistent within the theory. When afterwards a theory solver will determine that there is a conflict, the SAT solver can backtrack to the last propagation that was valid with the new added clause, instead of solving the whole problem from scratch. This method is not restricted to the algorithm itself, but can be addressed from an algorithm or user. That allows the user to solve a complicated SMT problem, then change a minor thing, e.g. drop a clause and add another one instead. With the incremental technique this second problem is far more easier, since the solver

can backtrack to the last propagation that is valid for the new clause, and start from this point, instead of solving the problem again. The authors of [39] did a comparative study in which they came to the result that incremental solving is in most cases much faster compared to cache-based approaches.

2.3.4.2 *Resolution Proofs*

The conflicts derived from the theory solver are usually optimized in the way that the number of literals used is minimized but still the unsatisfiability of the theory is maintained. The subset of the conflict is therefore often called *minimal infeasible subset*.

Definition 12 (Resolution Proof)**.**
Let $S = \{c_1, \ldots, c_t\}$ *be a set of clauses.* $P = (V_P, E_P)$ *is a directed acyclic graph partitioned into inner nodes and leaves.* P *is a resolution proof of the unsatisfiability of* $c_1 \wedge \cdots \wedge c_t$ *in* $\mathcal{LA}(\mathbb{Q})$*, if*

1. *each leaf in* P *is either a clause in* S *or a* $\mathcal{LA}(\mathbb{Q})$*-lemma (corresponding to some* $\mathcal{LA}(\mathbb{Q})$*-conflict* η*);*
2. *each inner node* v *in* P *has exactly two parents* v^R *and* v^L*, such that* v^R *and* v^L *share a common variable* p (pivot *variable) in the way that* $p \in v^L$ *and* $\neg p \in v^R$*. We derive* v *by computing the conjunction of* $v^R \wedge v^L$*;*
3. *the unique root node* r *in* P *is the empty clause.*

Let $S = \{c_1, \ldots, c_t\}$ be an $\mathcal{LA}(\mathbb{Q})$-unsatisfiable set of clauses, (A, B) a disjoint partition of S, and P a proof for the unsatisfiability of S in $\mathcal{LA}(\mathbb{Q})$. Then an interpolant I for (A, B) can be constructed by the following procedure [42]:
The unique root node r_I then represents the formula of an interpolant of A and B. Several methods are known to construct $\mathcal{LA}(\mathbb{Q})$-interpolants, e.g. [47], all have in common that they construct a single linear constraint based on the convex regions given to the function. One basic idea is to use Farkas' Lemma to construct a linear constraint separating these convex regions computed by linear programming. If those calls are handled separately, as in Algorithm 1, this will add for each $\mathcal{LA}(\mathbb{Q})$-conflict in the proof a linear constraint with a fixed normal [49] to the interpolant, and therefore will lead to a substantial blow-up in complexity.

2.4 MODEL CHECKING

Since our problem occurs in the context of a specific model-checking algorithm, we want to give a very short overview of the overall algorithm. For a detailed description we refer to [4]. We will skip most of the formal definitions and only give an overview. The system

Algorithm 1 : Construct an interpolant from a resolution proof

begin
  $I = P$
  for *every leaf* $\nu_P \in P$ *associated with a clause in* S **do**
    if $\nu_P \in A$ **then**
      $\nu_I = \nu_P \downarrow B$
    else if $\nu_P \in B$ **then**
      $\nu_I = \top$
  for *every leaf* $\nu_P \in P$ *associated with a* $\mathcal{LA}(\mathbb{Q})$*-conflict* η **do**
    $\nu_I = \mathtt{Interpolant}(\eta \setminus B, \eta \downarrow B)$
  for *ever inner node* $\nu_P \in P$ **do**
    if $\nu_P \notin B$ **then**
      $\nu_I = \nu_I^L \vee \nu_I^R$
    else if $\nu_P \in B$ **then**
      $\nu_I = \nu_I^L \wedge \nu_I^R$
  return I

model is based on a Linear Hybrid Automaton Extended with Discrete States (LHA+D), which is based on a finite set of Boolean and continuous variables. It alternates between two transitions, so called *flows* that are continuous transitions, and *jumps* that are discrete transitions. The set of variables is usually separated into four different sets, but for our purpose we can just consider them split in three. A set of continuous variables C, and two sets of Boolean variables D, M, where D consists of variables that are used to model something, e.g. counters, and M consists of so called mode variables that encode the discrete states of a traditional linear hybrid automaton. Each discrete transition (jump) is a guarded assignment, which is modeled as an SMT formula over rational linear arithmetic and change the variables $x_i \in D \cup M \cup C$ to values that satisfy either linear or Boolean expressions. Additionally there are SMT formulas, called *global constraints*, which specify lower and upper bounds for continuous variables, possibly for every mode individually.

Definition 13 (The Semantics of a LHA+D without formal definition)**.**

- *A* state *of an* LHA+D *is a valuation* $s = (d, c, m)$ *of* D, C *and* M*.*
- *There exist conditions for a continuous transition from a state* $s^i = (d^i, c^i, m^i)$ *to a state* $s^{i+1} = (d^{i+1}, c^{i+1}, m^{i+1})$*, which mainly involves that on the way from* s^i *to* s^{i+1} *neither a transition is triggered nor the global constraints are violated.*
- *A finite sequence of states* $\left(s^i = \left(d^i, c^i, m^i\right)\right)_{0 \leqslant i \leqslant n}$ *is a trajectory of an* LHA+D*. All states satisfy the global constraints, each transition*

satisfies its guarded assignment, and its predecessor was of a specific transition type.

- *A state* $s' = (d', c', m')$ *is reachable from the state* $s = (d, c, m)$*, if there exists a trajectory that starts in* s *and ends in* s'*. For a set of initial states* Init *we call the set that consists of states that is reachable from a state in* Init *the* reachable state set*.*

The LHA+D is called safe for the states set Init and Unsafe, if the reachable state sets of Init and Unsafe are disjoint. The algorithm performs a backward fixpoint computation. Starting with Unsafe the algorithm repeatedly computes the pre-image until it reaches a fixpoint or halts when an intersection with Init is detected.

2.4.1 *Representation of State Sets*

To represent a state set the algorithm needs to handle huge (quantifier free) SMT formulas with rational linear arithmetic. Therefore it represents every linear constraint present in the formula by a Boolean variable, identical to the procedure that was described in Section 2.3.4. This leads to a Boolean structure, which is then represented by a Functionally Reduced AND-Inverter Graph (FRAIGS), which are basically AND-Inverter Graphs (AIGS) where no two nodes exist that represent the same Boolean formula. Since the previous substitution enriches the AIGS with linear constraints, the structure is called LINAIGs. Figure 2 shows the structure of a LINAIG.

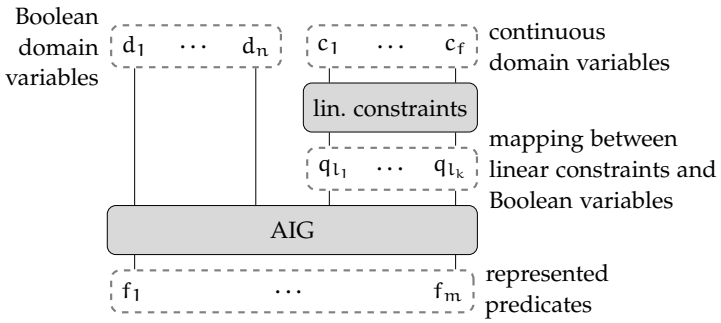


Figure 2: Structure of LINAIGs, which combine Boolean and continuous variables to model state sets of a hybrid linear automaton.

2.4.2 *Computation of Continuous Steps*

The computation of the continuous steps is basically a quantified SMT formula. Since the LINAIGSs are not designed to handle quantifiers, the quantifiers have to be eliminated. To avoid a conversion of a LINAIG to CNF or Disjunctive Normal Form (DNF), the quantifier elimination is done by Loos's and Weispfennig's test point method

[40] instead of Fourier-Motzkin algorithm. With this method, universal quantifiers are converted to finite conjunctions and existential quantifiers into finite disjunctions. The subformulas used in both of them are derived from the original formula by suitable test-points, which are again formulas of linear arithmetic. The number of test-points necessary for this procedure is linearly correlated to the number of linear constraints in the original formula. Thus for each elimination of a single real-valued variable, the number of linear constraints potentially grows quadratic with the number of linear constraints that are used in the original formula.
To enhance the performance of the algorithm it combines the two approaches *onioning*, and CEGAR.

2.4.3 *Onioning*

The idea behind this approach is that in the computation of the fixpoint we repeatedly compute a pre-image of the current state set. The state set itself therefore can be written as a union of pre-images of a specific depth in the fixpoint computation. To calculate the next pre-image it is not necessary to compute the pre-image of every single former pre-image, since only the states that are added due to the last pre-image will produce possibly new states. This idea was introduced in the late eighties in the context of symbolic model-checking in [15]. It is however not clear if the computation of the pre-images is easier on a smaller set, but combined with other methods this creates freedom to create simpler state sets to compute the pre-image.

2.4.4 *Counterexample-Guided Abstraction Refinement*

The algorithm will try to overapproximate some intermediate state set, and computes the pre-image of this, instead of the real state set. If a fixpoint is reached without a detection of an intersection with $\mathtt{Init}$ the model is safe. On the other hand an intersection with $\mathtt{Init}$ does not necessarily lead to $\mathtt{Unsafe}$, since the overapproximation could be the reason. With this in mind, the algorithm searches for a spurious counterexample, i.e. a trajectory from a state in the intersection to a state that is only in a state set of an overapproximation and not of the original intermediate state set. These trajectories are then used to refine the overapproximation, and the algorithm restarts from this point with a new computed overapproximation. The model is still unsafe, when a trajectory is found that leads to $\mathtt{Unsafe}$ that is always in the original intermediate state set. The algorithm terminates if such a real counterexample is found, or if a fixpoint is reached without a detection of an intersection with $\mathtt{Init}$.

2.4.5 *Epsilon Bloating*

To guide the overapproximation of the intermediate state set and forbid a huge overapproximation, [4] introduced the concept of epsilon bloating. For a given state set representation as an SMT formula $\Phi(x)$, where x are the continuous variables, we compute a bloated state set $\mathtt{bloat}(\Phi(x))$ by the following formula,

$$\mathtt{bloat}(\Phi(x)) = \exists v(W_\varepsilon(v) \wedge \Phi(x - v)) \ , \tag{8}$$

where v is a vector of real variables with the size of x, and $W_\varepsilon(v)$ defines a lower and upper bound for every variable v_i. Previously we defined an ε_i for every variable. $W_\varepsilon(v)$ is defined as

$$W_\varepsilon(v) = \bigwedge_i -\varepsilon_i (u_i - l_i) \leqslant v_i \leqslant \varepsilon_i (u_i - l_i) \ , \tag{9}$$

where l_i, u_i are the lower bound and upper bound of the variable x_i. In the current standard setting of the algorithm every ε_i is set to a global ε, such that the range of each variable is only defined by the size of the bounding box of the variable.

Remark The given state set $\Phi(x)$ is always a subset of $\mathtt{bloat}(\Phi(x))$, since $v = 0$ is valid for every W_ε.

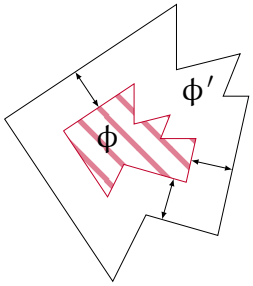


Figure 3: An ε-bloated state set

2.4.6 *Resume*

We therefore have a model-checking algorithm that searches for a fixpoint by a backward approach, so instead of searching for all states that are reachable from $\mathtt{Init}$, it searches for all states that can reach $\mathtt{Unsafe}$ and therefore would also lead to $\mathtt{Unsafe}$. The computational effort for the pre-image computation is linearly correlated to the number of linear constraints, and leads to a potentially quadratic growth of linear constraints used in the formula of the pre-image. Therefore the pre-image computation itself will potentially grow quadratically. With the two approaches CEGAR and onioning, the algorithm has two possibilities where the state set can be overapproximated. Regardless

of which situation, the quadratic growth of linear constraints in the pre-image computation and the fact that the elimination of quantifiers has a quadratic factor in its computation time depending on the number of linear constraints, the goal of a simple representation is to lower the amount of linear constraints. This leads us to our problem definition in the next chapter.

3

PROBLEM DEFINITION

In this chapter we will give a formal definition of the problem to *minimize interpolants*. In the previous Section 2.4 we introduced the algorithm in which the problem occurs. The main goal is a *simple* representation of a state set that should have fewer linear constraints. In both concepts to improve the algorithm CEGAR and onioning, there are two state sets. In case of CEGAR there is the intermediate state set Φ and the one computed by the epsilon bloating $\mathtt{bloat}(\Phi)$. In case of onioning there is the set of all with i iterations reachable points Φ_i and the state set that is only needed to compute the next pre-image $\Phi_i \setminus \Phi_{i-1}$. It is also done that these two are again combined, such that in CEGAR not the actual reachable set is bloated, but $\Phi_i \setminus \Phi_{i-1}$ the state set that is actual needed for the next pre-image computation. Such that the two state sets in this situation are $\Phi_i \setminus \Phi_{i-1}$, and $\mathtt{bloat}(\Phi_i \setminus \Phi_{i-1}) \cup \Phi_i$.
All these cases have in common that there are two state sets, where one is a subset of the other, e.g. $\Phi \subseteq \mathtt{bloat}(\Phi)$. In this case there is a well known theorem of mathematical logic that introduces the concept of an (Craig) interpolant:

Definition 14 (Craig interpolant [16])**.**
Let A *and* B *be two formulas, such that* $A \wedge B \models \bot$. *A* Craig interpolant I *is a formula such that*

1. $A \models I$
2. $I \wedge B \models \bot$
3. *the uninterpreted symbols in* I *occur both in* A *and* B, *the free variables in* I *can occur freely both in* A *and* B.

Remark *We will sometimes use only the concept of an interpolant, not a Craig interpolant, which does not explicit enforce the third point of the definition.*

Remark *Sometimes an interpolant is also defined for formulas, such that* $A \models B$, *instead of* $A \wedge B \models \bot$. *In this situation* $\neg B$ *will suffice the condition for the definition,* $A \wedge \neg B \models \bot$.
The second condition $I \wedge B \models \bot$, *can also be reformulated such that* $\neg I \models B$.

With this concept, we can formulate our problem by the following:

Definition 15 (Minimize Interpolant Problem)**.**
Given two state sets $A, B \subset \mathbb{B}^n \times \mathbb{R}^m$. *Let* $\mathcal{I}$ *be the set of all interpolants for* A *and* B. *The solution for the* Minimize Interpolant Problem I^* *is an interpolant out of* $\mathcal{I}$ *that has minimal number of constraints.*

This problem is at least NP-hard, which can be shown by a reduction of k-*polyhedral separability*. Given two sets of points, does there exist a set of k hyperplanes, such that every pair out of these two sets is separable? This problem is NP-complete.

Lemma 16.
The Minimize Interpolant Problem *is NP-hard.*

Proof. To reduce k-polyhedral separability problem to our problem we show that every instance of the k-polyhedral separability problem can be solved by an algorithm solving the *Minimize Interpolant Problem*. Therefore we just re-formulate every point as a convex region. The disjunction of all these convex regions of one set will give a suitable formula for our problem. Therefore an algorithm for the *Minimize Interpolant Problem* will directly answer the k-polyhedral separability problem. If the instance of the k-polyhedral problem is separable by at most k hyperplanes, the k-bounded interpolant problem is solvable. If it were not solvable there could not exist k hyperplanes that separates the set of points. And if the formulated k-bounded interpolant problem is solvable, the k-polyhedral separability problem is solvable by the same formula. □

Example 2.
Figure 4 shows two state sets A, B*, with no Boolean complexity, i.e.* $n = 0$*. One representation of those sets is*

$$A = (l_1 \wedge l_2) \vee l_3 \vee l_4 \qquad \textit{and} \qquad B = (l_5 \wedge l_6) \vee (l_7 \wedge l_8)$$

In this case, A *itself is an interpolant of* A *and* B*.*

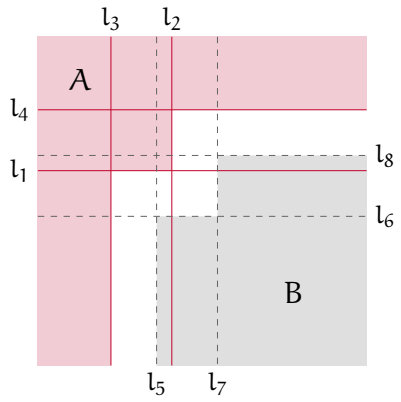


Figure 4: Running example for our algorithm

4

RELATED WORK

In this chapter we will give an overview over the techniques that are currently used to solve the *Minimize Interpolant Problem*, Definition 15. Every algorithm we present here solves the problem only heuristically. First we will present the method of *Redundancy Removal* that will just remove so called *redundant* constraints, i.e. they will construct another description for A with a subset of linear constraints already used in the description of A. The extended approach *Constraint Minimization* is to find an interpolant for A and B by defining *Don't Care*-Sets and removing more constraints as long as the over-approximation of A only extends in these Sets.
Beautiful Interpolants [2] is a method that constructs an interpolant from scratch. It iteratively searches for points that are not separated, and refining the interpolant step by step. The last approach *Simple Interpolants for Linear Arithmetic* [49] constructs the interpolant by simplifying the resolution proof given by an SMT solver.

4.1 REDUNDANCY REMOVAL AND CONSTRAINT MINIMIZATION

In this section we will first present a method that reduces the number of linear constraint in the description of a state set by removing all *redundant* constraints. This is not as straightforward as in the situation of convex polyhedra. A linear constraint l is redundant for a formula F if there exists a formula G that depends on the same linear constraints except for l and F and G represents the same predicates, formally:

Definition 17 (Redundancy of linear constraints)**.**
Let F *be an* SMT *formula over Boolean variables* $b_1, \ldots, b_k$ *and linear constraints* $l_1, \ldots, l_n$ *over the real-valued variables* $c_1, \ldots, c_f$*. The linear constraints* $l_1, \ldots l_r (1 \leqslant r \leqslant n)$ *are called* redundant *in the representation of* $F(b_1, \ldots, b_k, l_1, \ldots, l_n)$ *if and only if there is a Boolean formula* G *with the property that* $F(b_1, \ldots, b_k, l_1, \ldots, l_n)$ *and* $G(b_1, \ldots, b_k, l_{r+1}, \ldots, l_n)$ *represents the same predicate.*

To check for redundant constraints the algorithm will make a copy c_i' for every real-valued variable c_i. Then they formulate every linear constraint l_j over the copied variables and denote this constraint with l_j'. The check for redundancy is then based on the following theorem [19]:

Theorem 18 (Redundancy check)**.**
The linear constraints $l_1, \dots l_r (1 \leqslant r \leqslant n)$ *are redundant in the representation of* $F(b_1, \dots, b_k, l_1, \dots, l_n)$ *if and only if the predicate*

$$F(b_1, \dots, b_k, l_1, \dots, l_n) \oplus F(b_1, \dots, b_k, l'_1, \dots, l'_n) \wedge \bigwedge_{i=r+1}^{n} (l_i \equiv l'_i) \quad (10)$$

(where $\oplus$ *denotes exclusive-or) is not satisfiable by any assignment of Boolean values to* $b_1, \dots, b_k$ *and real values to the variables to* $c_1, \dots, c_f, c'_1, \dots, c'_f$*.*

The proof of this theorem is given in [48] and is constructive. To achieve this they introduce a Don't Care Set (DC) for F, which consists of all Boolean configurations which are not $\mathcal{LA}(\mathbb{Q})$-sufficient, and then efficiently construct G, which holds $F = G$ for every configuration not in DC. Formally the DC is defined by

Definition 19 (Don't Care Set)**.**
The Don't Care Set DC *induced by linear constraints* $l_1, \dots, l_n$ *is defined as*

$$\begin{aligned} DC := \{(v_{l_1}, \dots, v_{l_n}) \mid\ & (v_{l_1}, \dots, v_{l_n}) \in \{0,1\}^n \text{ and} \\ & \forall (v_{c_1}, \dots, v_{c_f}) \in \mathbb{R}^f\ \exists 1 \leqslant i \leqslant n \text{ with} \\ & l_i (v_{c_1}, \dots, v_{c_f}) \neq v_{l_i}\}. \end{aligned}$$

This is useful, since common SMT solvers (with the option of minimizing conflict clauses) will produce a representation of a subset of DC as a byproduct of the solution of formula (10). The new formula G can be computed in linear time from a resolution proof of a formula in CNF. As mentioned in Section 2.3.4 resolution proofs are available in any modern SMT solver, if the user turns on proof logging.
With this approach and this definition of the DC set, the representation would remove redundant constraints, but it would not overapproximate the state set. However the method was expanded in [18] to compute interpolants of A and B. Therefore they extend the DC set from Definition 19 by the Boolean combinations that are representing the region $\neg A \wedge \neg B$. With this extension of the set DC they allow to differ not only in Boolean configuration that are not $\mathcal{LA}(\mathbb{Q})$-sufficient, but also in regions that are neither in A nor in B. This turns the method of *Redundancy Removal* in a method of interpolation, named *Constraint Minimization*. The main difference is that *Constraint Minimization* will overapproximate the state set A, when *Redundancy Removal* would only change the representation.
For both approaches the fundamental disadvantage is that they will only use linear constraints already known to the system.

4.2 BEAUTIFUL INTERPOLANTS

This method was introduced in [2]. The following algorithm is initially not designed to handle state sets with additional Boolean com-

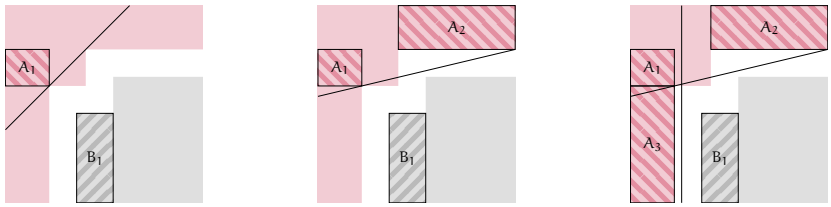
plexity. Since this is a main part of our problem we have to redefine certain steps to handle this issue. It will construct an interpolant of two state-sets from scratch, i.e. without the usage of a previously computed interpolant.
The basic idea is to search for a model that is not yet correctly separated. Expand this model to a sample, i.e. a convex region, and further try to partition these samples into a small amount of samplesets, such that each A-sampleset can be separated from each B-sampleset by a single linear constraint. Therefore the main objects of the algorithm are the two sets of samplesets S_A and S_B, and the partial interpolant map PItp. PItp maps pairs of sets of convex regions to a hyperplane, i.e. it maps $(s_A, s_B) \in S_A \times S_B$ to a hyperplane that will separate every convex region in s_A from every convex region in s_B.
The authors introduce three different kinds of subroutines for the algorithm, which are all formulated as guarded commands.

1.) Rules that distribute samples into samplesets, i.e. MERGE and SPLIT. Where MERGE and SPLIT are contrary commands, where MERGE will reduce and SPLIT will increase the number of samplesets by one.

2.) Rules that check interpolants and construct partial interpolants, i.e. CHECKITPA and CHECKITPB will check if there exists a model in A or B that is not yet separated, and PARTIALITP a command that will fill a single entry of the partial interpolant map PItp for a pair of sampleset if it is possible.

3.) Rules that check for termination.

Note that invariantly, if (s_A, s_B) is in the domain of PItp then the linear constraint $\mathrm{PItp}(s_A, s_B)$ is an interpolant for $(\bigvee s_A, \bigvee s_B)$.
In the initialization the sets S_A, S_B and the map PItp are set to be empty. Then the algorithm would construct an interpolant C out of PItp and check whether $A \wedge \neg C$ is satisfiable, alternatively it would check if $B \wedge C$ is satisfiable. In either case the found model in A (or B) is expanded to a sample, and put in S_A (or S_B) as a new sampleset. If the number of samplesets of A is bigger then two, the algorithm has now the choice to MERGE the new convex region into another set $s_A \in S_A$. Either way there is a new A-sampleset and therefore the algorithm has to execute PARTIALITP for every existing B-sampleset and compute a new linear constraint for each of them. If this is not possible for a single pair, the algorithm has to execute a SPLIT. How this is exactly guided is not well described in the paper, they mention to use the Farkas coefficients to choice a split, but if they for example split after a single partial interpolant failed, or if they use multiple failed PARTIALITP-calls to find a choice that is suitable for multiple splits is not known. It is important to notice that not every split will fix the problem, and perhaps multiple splits are needed. To understand



(a) (b) (c)

Figure 5: Running example with the technique of *Beautiful Interpolants*

the behavior of this algorithm better, we show what could happen in our example.

Example 3.
In Figure 5 (a) two points a_1 *and* b_1 *are enlarged to the shaded convex regions* A_1, B_1*, then* $S_A = \{\{A_1\}\}$ *and* $S_B = \{\{B_1\}\}$*.*
In (b) we find a point a_2 *which is on the wrong side of the interpolant and therefore the convex region* A_2 *is introduced, i.e.* $S_A = \{\{A_1\},\{A_2\}\}$*. Now the algorithm takes the opportunity to merge these sets to one, such that* $S_A = \{\{A_1, A_2\}\}$*. In this case we have to update every partial interpolant already computed for* $\{A_1\} \in S_A$*. In (c) we find a third point* a_3*, but after we also merged* A_3 *to the other sets, the update for the partial interpolants returns an error. This means we have to split the set* $\{A_1, A_2, A_3\}$ *somehow, e.g. to* $\{A_1, A_2\}$ *and* $\{A_3\}$*.*
Recognize that not every choice of the split will end in direct solvable solution, e.g. if we would split into $\{A_1\}$ *and* $\{A_2, A_3\}$ *it would return the same error and would force another* Split*.*

To ensure the termination of the algorithm they remember every set that enforced a Split in the process and will never perform a Merge that end up in such a set. Since the number of sets are finite, this leads to a limited number of possible sets and therefore to its termination. The interpolant build out of the partial interpolant is defined by

$$\textsc{Cand}\,(S_A, S_B, \mathrm{PItp}) \equiv \bigvee_{s_A \in S_A} \left(\bigwedge_{s_B \in S_B} \mathrm{PItp}(s_A, s_B) \right) . \tag{11}$$

Remark In a nutshell this method efficiently produces small interpolants, when the situation is trivial, i.e. the state sets are separable with one or two linear constraints. It has to terminate to actually have an interpolant to work with, and the only way to speed up the process is to skip merges. This on the other hand leaves the algorithm to an enumeration of convex regions and separating every single pair of them on their own. If someone wants to implement this approach

there is a fair amount of unknown decisions, e.g. which convex region they choose in a split, in which order they execute the subroutines when multiple guards are fulfilled. Last but not least the interpolant build by this method is always in disjunctive normal form, which can lead to exponentially larger form than needed. Since And-Inverter-Graph (AIG) or Linear-And-Inverter-Graph (LINAIG) are known to be a better representation for Boolean formulas, the interpolant must be transformed to actually work with it.

As mentioned in the beginning this method was introduced with no Boolean complexity in the state sets in mind. First of all the samples are specific to a Boolean assignment, in the step where they enlarge the sample to a convex region, they have now the option to find a more general formula for the Boolean assignment as well. The basic condition they use in the actual construction of the interpolant is that for each pair of samplesets s_A, s_B the partial interpolant map provides an interpolant for $\bigvee s_B$ and $\bigvee s_A$. Assume they hold on to that, then the partial interpolant is no longer a formula only over the real variables (a linear constraint), but over the Boolean variables that are used in the description of the state sets as well.
This would lead to a different function to compute the partial interpolants. One possible way would be to search for a linear constraint that still separates all samples s_A from s_B. If there are no Boolean assignments of a sample in s_A equal to any sample in s_B the samplesets have to be separated due to the Boolean variables, i.e. a Boolean interpolant. All this could lead to the fact that the method has to enumerate over all Boolean assignments.

4.3 SIMPLE INTERPOLANTS FOR LINEAR ARITHMETIC

In Section 2.3.4.2 we introduced a way to compute interpolants directly as a result of a resolution proof. The disadvantage we already mentioned is that they typically handle the separation of the convex regions separately, where each of them add a linear constraint with a fixed normal to the interpolant. This leads to a substantial blow-up in the number of linear constraints. The idea behind this method is therefore to relax the $\mathcal{LA}(\mathbb{Q})$-conflicts to gain more freedom in these conflicts and then try to find constraints that separate more than one conflict. The method is based on a lemma in [49]:

Lemma 20.
Let P *be a proof of* $\mathcal{LA}(\mathbb{Q})$*-unsatisfiability of* $A \wedge B$*, let* $\neg \eta$ *be an* $\mathcal{LA}(\mathbb{Q})$*-lemma in* P *not containing literal* $\neg l$*, and let* l *be implied for* A*, i.e.* $A \models l$ *and* l *does not occur in* B*. Then Craig interpolation according to [42] applied to* P *with* $\neg \eta$ *replaced by* $\neg \eta \vee \neg l$ *computes a Craig interpolant for* A *and* B*.*

They are therefore free to extend the conflicts with every negation of implied literals for A or B while keeping the interpolant correct. Additionally they introduce a system of linear constraints that will compute a shared interpolant for multiple conflicts. Assume a fixed set of conflicts $\{\eta_1, \ldots, \eta_r\}$. Each of them defines a convex region by a conjunction of linear constraints of A and B, i.e. $\eta \setminus B$ for A and $\eta \downarrow B$ for B, as we described it in 2.3.4.2. For the conflict η_j we write $A_j x \leqslant a_j$ for the system $\eta_j \setminus B$, and $B_j x \leqslant b_j$ for the system derived by $\eta_j \downarrow B$. They introduce a system of inequations

$$\lambda_j^T A_j + \mu_j^T B_j = 0^T \tag{12}$$
$$\lambda_j^T A_j = i^T \tag{13}$$
$$\lambda_j^T a_j \leqslant \delta \tag{14}$$
$$\delta + \mu_j^T b_j \leqslant -1 \tag{15}$$
$$\lambda_j^T \geqslant 0 \tag{16}$$
$$\mu_j^T \geqslant 0\,, \tag{17}$$

where for each solution the variables i and δ define a single inequation $i^T x \leqslant \delta$, which separates all conflicts simultaneously. This is solved by linear programming, without a objective function.

Remark Since they do not have an objective function, the problem is never unbounded, it is feasible or unfeasible.
They also exclude the solution where everything is set to zero naturally by the equation (15).
The downside to that approach is that they can not handle strict inequalities properly, since they will never allow that the convex regions touches each other in any point, due to the equation (15).

These ideas combined lead to the following algorithm.
They first compute a resolution proof for A and B. Then they extend the conflicts by literals computed after a scheme called Lemma Localization, introduced in [46]. After that they compute an undirected compatibility graph $G = (V, E)$, where V consists of all conflicts, and there is an edge (η_i, η_j) if there is a shared interpolant of η_i and η_j, i.e. the system (12-17) is feasible. Their first idea was a greedy algorithm, which iteratively tries to extend a shared interpolant by a conflict that is connected to the other conflicts already in the shared interpolant. If this is possible they extend the interpolant, otherwise they will try another conflict. When no conflict can extend the interpolant, they create a new interpolant by an edge of the graph, where both conflicts are not used in a previously shared interpolant.
The second idea was to iteratively find the maximum subset of conflicts in the graph. After finding a maximum subset, they would remove all conflicts from the subset in the graph. They implemented this idea by formulating the maximal subset as an SMT-formula. We will not go into more details here, since their experimental results

showed that the greedy algorithm performs better and there were few examples where the maximum subset method would find smaller interpolants.

5

THE ALGORITHM FOR IMPROVING INTERPOLANTS WITH LINEAR ARITHMETIC

In this chapter we will present our heuristic method to compute good solutions for the *Minimize Interpolant Problem*. A detailed description of the problem is given in Chapter 3.
Our algorithm is based on an extension of the following proposition.

Proposition 21 (Prop. 3 in [43])**.**
The hyperplanes $L_1, \ldots, L_h$ *separate the sets (of points)* P *and* Q *in the sense of* h*-polyhedral separability through some Boolean formula if and only if for every pair of points,* $p \in P$ *and* $q \in Q$*, there exists an* i $(1 \leqslant i \leqslant h)$ *such that* p *and* q *lie on different sides of the hyperplane* L_i*.*

We will extend this proposition to our context in two ways. First by changing the sets of points P and Q to state sets. Therefore they can be viewed as a union of convex regions in different Boolean spaces. Second we will change the condition appropriate to the situation by forcing an existence of such an hyperplane only in the case where they are in the same Boolean space.

Proposition 22 (Extension of Proposition 21)**.**
The set of hyperplanes $L = \{L_1, \ldots, L_h\}$ *separates the state sets* A *and* B *through some Boolean formula only over the Boolean variables used in the state sets and the variables induced by the hyperplanes in* L *if and only if for every pair of points* $p \in A$ *and* $q \in B$ *with* $p_{\mathbb{B}} = q_{\mathbb{B}}$ *there exists an* i $(1 \leqslant i \leqslant h)$ *such that* p *and* q *lie on different sides of the hyperplane* L_i*.*

The proof of the extension is basically the same as the one given in [43] despite two differences caused by our changes. For the sense of completeness we will give the complete proof with the adjustments.

Proof. Let $I = \{1, \ldots, h\}$ be the index set of hyperplanes, and $J = \{1, \ldots, n\}$ the index set of Boolean variables that describe the Boolean space of the state sets. We denote the induced variables of a hyperplane $L_i \in L$ with ξ_i, and the Boolean variables of the description of the state sets with $\vartheta_1, \ldots, \vartheta_n$. First we can rewrite the state sets as mentioned above, such that for every Boolean space we have a representation of the state set A as a union of convex sets. Analogously, we find for every Boolean space a representation of B with convex sets. Now we can refine those representations by partitioning every convex set by the hyperplanes in L. For every convex set C in this refined representation and for every constraint $l \in L$ all points in C are on the same side of the hyperplane l, since otherwise it is a contradiction that C is a convex set of the refined representation. If

two points lie on the same side of each hyperplane the Boolean values of the functions induced by the hyperplanes are equal. If L separates the state sets A and B in the sense of the definition then for any two points $p \in A$ and $q \in B$ there is a variable that has a different truth value at p and q. If they are in the same Boolean space, i.e. the points are equal under all variables ϑ_j, there exists at least one variable ξ_i with different truth values for q and p. This implies that at least the hyperplane L_i separates the two points.
For the other direction we suppose that for every Boolean space $\mathcal{B}_k$ every two points p_i^k, q_j^k are separated by at least one hyperplane. Let ζ be any point in $A \cup B$, and split the set of indices I and J such that for every $i \in I' \subset I$ and $j \in J' \subset J$ the variables ξ_i and ϑ_j are true for the point ζ. Otherwise for every point $i \in I \setminus I'$ and $j \in J \setminus J'$ the variables ξ_i and ϑ_j are false for the point ζ. Let

$$\phi_\zeta = \phi_\zeta(\xi_1, \ldots, \xi_h, \vartheta_1, \ldots, \vartheta_n) = \bigwedge_{i \in I'} \xi_i \bigwedge_{j \in J'} \vartheta_j \bigwedge_{i \in I \setminus I'} \overline{\xi_i} \bigwedge_{j \in J \setminus J'} \overline{\vartheta_j}$$

denote a Boolean formula associated with ζ. Then ϕ_ζ is true at ζ. Consider the formula

$$\phi = \bigvee_{p \in A} \phi_p \ .$$

Then for every point in A the formula ϕ is true. Otherwise for every point $q \in B$, ϕ_p is false for q for every $p \in A$, since at least one variable has another truth value at the two points. Thus ϕ separates A and B with only the usage of the Boolean variables stated in the condition. □

We use this proposition in our algorithm as follows. The algorithm iteratively improves the set L by replacing two linear constraints by one, while preserving the invariant that L satisfy Proposition 22. The linear constraints in L are called *interpolant constraints*. Every new constraint added to L can be described separately by a linear combination of constraints in A and B. This leads to the fact that if the initial L only contains constraints described over local variables of A and B, the output of our algorithm only contains such constraints, for details we refer to the linear constraints of the LP (21) and (22) in Section 5.1.3. Hence, our algorithm is a local search heuristic.
In the following section, we describe the test whether two linear constraints can be replaced by one and in Section 5.2, we describe some techniques to improve the performance. In particular, we describe our approach to reduce the number of pairs of linear constraints that are tested. Most approaches are of a heuristic nature.
The interpolant itself is then constructed by using the method described in Section 4.1. We therefore deliver a sufficient set of constraints to construct the interpolant, so in the constraint minimization the part of finding redundant constraints can be skipped. This

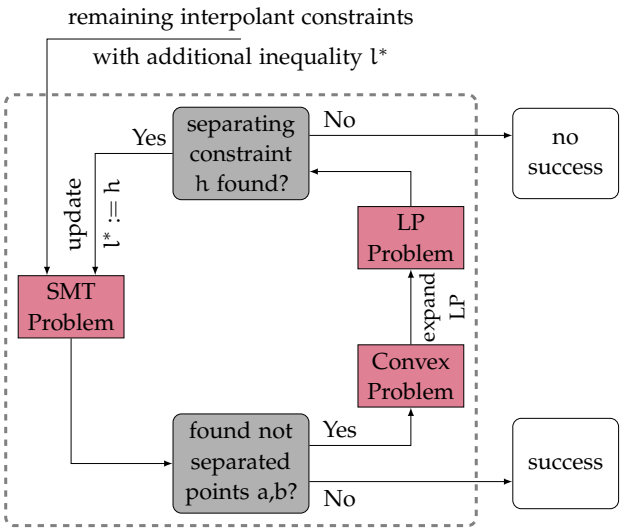
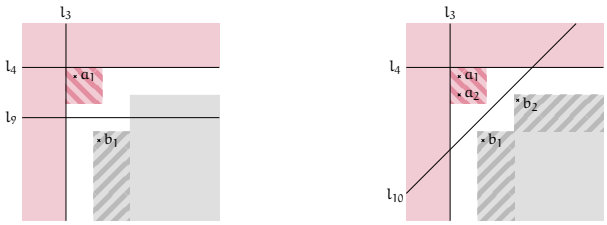


Figure 6: Sketch of the core part of the heuristic

together with the fact that all interpolant constraints are defined over local variables of A and B lead to the fact that the resulting interpolant is in fact a Craig interpolant.

5.1 TEST WHETHER ONE ADDITIONAL LINEAR CONSTRAINT IS SUFFICIENT

This is the core part of the heuristic, where we determine if we can reduce the size of the interpolant, i.e. the set of interpolant constraints. It will test if it is possible to substitute two given interpolant constraints by one new linear constraint l^*. Notice that when removing two interpolant constraints, there will be pairs $a \in A, b \in B$ of points with $a_{\mathbb{B}} = b_{\mathbb{B}}$ that can not be distinguished by the remaining interpolant constraints L, i.e. $l(a) = l(b)$ for all $l \in L$. We will test whether all those pairs of points can be distinguished by a single new linear constraint.
Basically, we iteratively collect such pairs of points and construct a linear constraint l^* separating all pairs of points already found, until either all pairs of points can be distinguished with the additional help of l^* or no such linear constraint can be found. Figure 6 gives a sketch of the algorithm.
In order to guarantee termination, it does not suffice to collect pairs of points, as there can be an infinite number. Therefore, we construct convex regions C_a and C_b around the points a and b described only by constraints known to the system, i.e. for C_a we only use linear constraints used in the description of A and additionally all remaining interpolant constraints. The convex sets are contained in the respective sets, i.e. $C_a \subseteq A$ and $C_b \subseteq B$. Since there are only a finite number of linear constraints in the description of the state sets and interpolant



(a) (b)

Figure 7: Example of reducing the set of interpolant constraints, still satisfying the conditions of Proposition 22

constraints, there are only a finite number of possible convex sets describable by these constraints. This leads to a termination of this part of the algorithm in a finite number of steps.
Hence, we need three sub-algorithms. One for finding pairs of points that can not be distinguished, one for constructing the convex regions around them, and one for constructing a linear constraint separating a given set of pairs of convex regions. An algorithm for the first problem is given in Section 5.1.1, which is based on solving an SMT-problem. A simple solution for the second problem is given in Section 5.1.2. Finally, we give a solution for the third problem based on linear programming in Section 5.1.3.

Example 4 (cont.)**.**
As mentioned in the previous part the set of linear constraints $L = \{l_1, \ldots, l_4\}$ *of* A *is a valid set of interpolant constraints, i.e. satisfying Proposition 22. Figures 7*(a) *and* (b) *show the solutions of the three sub-algorithms from the test, whether* l_1 *and* l_2 *can be substituted by a single new constraint. In the first iteration, shown in* (a) *we find a pair of points* a_1, b_1 *that are indistinguishable, once* $L = \{l_3, l_4\}$*. After that we compute convex regions around those points, shown as highlighted areas, and finally compute a linear constraint* l_9 *that separates this pair of convex regions.* $L \cup \{l_9\}$ *does not satisfy Proposition 22, therefore the algorithm starts a second iteration, shown in* (b)*, and finally computes a linear constraint* l_{10} *that simultaneously separates both pairs of convex regions found in the first and the current iteration. Finally* $L \cup \{l_{10}\}$ *satisfies Proposition 22 and therefore one constraint less is needed in the interpolant.*

5.1.1 *Finding Pairs of Indistinguishable Points with Satisfiability Modulo Theory*

Notice that we are in the situation that we have to test whether our tentative new linear constraints l^* together with the remaining linear constraints L of the interpolant suffice to construct an interpolant between state sets A and B. The linear constraints are not sufficient if

and only if we can find points $a \in A, b \in B$ with $a_{\mathbb{B}} = b_{\mathbb{B}}$ that are not distinguishable, i.e. for all $l \in L \cup \{l^*\}\, l(a) = l(b)$ holds.
To solve this by an SMT-solver we use the following formula:

$$(a \in A) \wedge (b \in B) \wedge (a_{\mathbb{B}} = b_{\mathbb{B}}) \wedge \left(\bigwedge_{l \in L \cup \{l^*\}} l(a) = l(b) \right) \tag{18}$$

Either we found a valid solution and therefore two points a, b that are not separable by any linear constraint in $L \cup \{l^*\}$ or we found that $L \cup \{l^*\}$ is a valid set of interpolant constraints. The problem has only minor changes in every iteration, because we only substitute the old condition $l^*(p) = l^*(q)$ with an updated linear constraint l^*. This gives us the opportunity to use the advantage of using an incremental SMT, described in Section 2.3.4.1.

5.1.2 *Enlarge Pairs of Indistinguishable Points to Convex Regions*

If we have found a pair of points a, b as described in the previous section, we want to find a convex region $C_a \subseteq A$ around a that is preferably large, such that no point in C_a can be distinguished from b and vice versa.
We achieve this by computing a convex region around a, such that all points within C_a are equal with respect to all linear constraints in $L \setminus \{l^*\}$ and all linear constraints used in the description of A.
For computing C_a we go through every linear constraint l that is used in the description of A, and test if a satisfies this linear constraint, i.e. we compute $l(a)$. We therefore collect linear constraints in a set C. Then l is added to C when $l(a)$ is *true*, or $\neg l$ if $l(a)$ is *false*. After evaluating this for every linear constraint in A, we compute the same for every interpolant constraint $l \in L \setminus l^*$. We do not use the current l^* in the description of the convex region, since we change this constraint in the next step, where we search for a new candidate. C_a is then computed as a conjunction of all constraints in C. As a remark we want to point out that C_a is not the largest convex region defined with these linear constraints including a. To enlarge these convex regions we can use more detailed information of the state sets, i.e. [4] provides a method to compute *polarities* of linear constraints in a state set. The *polarity* of a linear constraint l in a state set has three possibilities. It is *positive* if only l is used as a Boolean variable in the description of the state set, *negative* if only the negation $\neg l$ is used, or *both* descriptions are used. For completeness there is the possibility that none of the variables are used, but then this linear constraint would be redundant, which is tested earlier. With this information we can enlarge the regions even further by only using Boolean variables whose polarities are used in the description of the state set. For the interpolant constraint we would still have to use both Boolean variables, since there is no such information at hand.

5.1.3 *Finding a Linear Constraint Separating Pairs of Convex Regions with Linear Programming*

We try to find a solution to this problem by constructing an LP where a part of the solution represents a separating linear constraint. The LP does not solve the problem in general, as it will fix the side for every convex region in the formulation. Typically we will put all convex regions of A on one side and the convex regions of B on the other side, but this is not necessarily their required place. Furthermore, it assumes that all inequalities in A and B are non-strict, i.e. convex regions are enlarged by their boundary. Both deficiencies are handled heuristically later. We use an LP solver that handles rational arithmetic as errors in the coefficients preventing the algorithm from termination since we could be forced to separate the same pair of convex regions multiple times, due to rounding errors.
The construction of the LP is similar to the one that computes the linear constraints in resolution proofs, hence based on Farkas' Lemma. We expanded the approach to separate multiple pairs of convex regions.
Therefore, we define the variables $d \in \mathbb{Q}^m$ and $d_0 \in \mathbb{Q}$ that describe the new linear constraint l^* in the form $d^T x \leqslant d_0$. Pairs of convex sets (A^i, B^i) for $i \in \{1, \ldots, k\}$, which are present in the k-th iteration of the LP-problem for finding a new constraint for one test described in Section 5.1, all constructed by the enlargement of points to convex regions described in Section 5.1.2. The constraint $d^T x \leqslant d_0$ is implied by A^i, if there is a non-negative linear combination of the inequalities of A^i leading to $d^T x \leqslant d_0$. Similarly, the constraint $d^T x > d_0$ is implied by B^i, if there is a positive ε such that there is a non-positive linear combination of the inequalities of B^i leading to $d^T x \geqslant d_0 + \varepsilon$. All constraints can easily be formulated as linear constraints.
Let A^i, B^i be the convex sets of the i-th iteration, constructed by s_{A^i}, respectively s_{B^i}, conjunctions of linear constraints. Then A^i is formally defined by

$$A^i = \left\{ x \in \mathbb{R}^m \middle| \mathcal{A}^i x \leqslant \alpha^i \right\} \qquad \text{, with } \mathcal{A}^i \in \mathbb{Q}^{m \times s_{A^i}} \tag{19}$$

and

$$B^i = \left\{ x \in \mathbb{R}^m \middle| \mathcal{B}^i x \leqslant \beta^i \right\} \qquad \text{, with } \mathcal{B}^i \in \mathbb{Q}^{m \times s_{B^i}} \,. \tag{20}$$

We additionally introduce for every iteration $i \in \{1, \ldots, k\}$ variables for the factors of the linear combinations. This means we have to add s_{A^i} variables λ^i for A and s_{B^i} variables μ^i for B.
We look for an inequality that maximizes a simple measure of the distance of the constructed inequality to the convex regions. We achieve this by subtracting ε from the positive convex combination of the inequalities from A^i for l, i.e. the convex combination leads to $d^T x \leqslant d_0 - \varepsilon$. As we can scale any LP-solution by an arbitrary positive scalar so far, we have to normalize the solution. Therefore, we

restrict the linear combination of one region to be a convex combination.
Hence, we obtain the following LP, where all linear constraints except (23) and (28) are introduced for all $i \in \{1, \dots, k\}$:

$$
\begin{aligned}
\max \quad & \varepsilon \\
\text{s.t.} \quad & (\mathcal{A}^i)^T \lambda^i = d && (21)\\
& (\mathcal{B}^i)^T \mu^i = d && (22)\\
& \sum \lambda^1 = 1 && (23)\\
& (\alpha^i)^T \lambda^i \leqslant d_0 - \varepsilon && (24)\\
& (\beta^i)^T \mu^i \geqslant d_0 + \varepsilon && (25)\\
& \lambda^i \geqslant 0 && (26)\\
& \mu^i \leqslant 0 && (27)\\
& \varepsilon \geqslant 0 && (28)
\end{aligned}
$$

Constraints (21) and (22) enforce that the direction of the new constraint, described by d, is representable by the linear constraint of every convex region. Conditions (24-28) verify that convex regions are on the correct side of l^*, i.e. as stated above every A^i of A and every B^i of B are on different sides. Condition (23) normalizes the solutions.

Lemma 23.
If the LP *(21-28) has a feasible solution* $(d, d_0, \lambda^i, \mu^i, \varepsilon)$ *with* $\varepsilon > 0$*, then the correspoding linear constraint* $l: \; d^T x \leqslant d_0$ *separates each pair of convex regions.*

Proof. Let $(d, d_0, \lambda^i, \mu^i, \varepsilon)$ be a feasible solution of the LP with $\varepsilon > 0$. Given an arbitrary point $a_i \in A^i$ and an arbitrary point $b_i \in B^i$, then it follows that

$$
\begin{aligned}
d^T a_i &= \left(\lambda^i\right)^T \mathcal{A}^i a_i && (29)\\
&\leqslant \left(\lambda^i\right)^T \alpha^i && (30)\\
&\leqslant d_0 - \varepsilon < d_0 \, . && (31)
\end{aligned}
$$

For an arbitrary point $b_i \in B^i$, it follows that

$$
\begin{aligned}
d^T b_i &= \left(\mu^i\right)^T \mathcal{B}^i a_i && (32)\\
&\geqslant \left(\mu^i\right)^T \beta^i && (33)\\
&\geqslant d_0 + \varepsilon > d_0 \, . && (34)
\end{aligned}
$$

The first equations (29) and (32) are concluded from (21) and (22). The inequality in (30) is a consequence of (19) and (26), the reason why the inequality of (33) is different comes from the fact that the negative variables from (27) combined with (20) will flip the inequality. The last conclusion is either by (24) or by (25). □

5.1.3.1 *Extension to Non-Closed Polyhedra*

The construction of the convex regions around the points, described in Section 5.1.2, leads to the fact that A^i, B^i are not like we described them in (19) and (20). The correct description is

$$A^i = \left\{ x \in \mathbb{R}^m \middle| \mathcal{A}^i_{\leqslant} x \leqslant \alpha^i_{\leqslant} \wedge \mathcal{A}^i_{<} x < \alpha^i_{<} \right\} \tag{35}$$

and

$$B^i = \left\{ x \in \mathbb{R}^m \middle| \mathcal{B}^i_{\leqslant} x \leqslant \beta^i_{\leqslant} \wedge \mathcal{B}^i_{<} x < \beta^i_{<} \right\} \tag{36}$$

Obviously Lemma 23 is also true in the case that some of the inequalities are strict, since $\varepsilon > 0$. But in the case where the LP (21-28) has only feasible solutions where $\varepsilon = 0$, we need another condition to conclude this statement. In this case there are convex regions that share the same boundary, and so for the separation it is crucial that some strict constraints used in the description have non-zero coefficients in the solution. The following proposition states when we have found a separating constraint in case of $\varepsilon = 0$.

Proposition 24.
Assume the LP *(21-28) has optimal value of* 0 *and let* $(d, d_0, \lambda^i, \mu^i, \varepsilon)$ *be a feasible solution.*

1. *If for all* $i \in \{1, \ldots, k\}$ *either* $(\beta^i)^T \mu^i - d_0 > 0$ *or there exists a strict inequality* $s \neq 0^m$ *in* $\mathcal{B}^i_{<}$ *with variable* $(\mu^i)_s$ *such that* $(\mu^i)_s < 0$, *then* $d^T x \leqslant d_0$ *separates the regions.*

2. *If for all* $i \in \{1, \ldots, k\}$ *either* $(\alpha^i)^T \lambda^i - d_0 < 0$ *or there exists a strict inequality* $s \neq 0^m$ *in* $\mathcal{A}^i_{<}$ *with variable* $(\lambda^i)_s$ *such that* $(\lambda^i)_s > 0$, *then* $d^T x < d_0$ *separates the regions.*

Proof. Review the proof of Lemma 23, there we were able to conclude the strict inequality for both regions at the last point, by just stating that $\varepsilon > 0$. This time we have to conclude in the first case that b_i and in the second case that a_i is strict. Here we will only give a proof for the first case, the second case is analog.
So let assume that for all $i \in \{1, \ldots, k\}$ either $(\beta^i)^T \mu^i - d_0 > 0$ or there exists a strict inequality $s \neq 0^m$ in $\mathcal{B}^i_{<}$ with variable $(\mu^i)_s$ such that $(\mu^i)_s < 0$, then $d^T x \leqslant d_0$ separates the regions. As a repetition, this part holds for any feasible solution of the LP,

$$\begin{aligned} d^T b_i &= \left(\mu^i\right)^T \mathcal{B}^i a_i && (37)\\ &\geqslant \left(\mu^i\right)^T \beta^i && (38)\\ &\geqslant d_0 \,. && (39) \end{aligned}$$

So let us assume for i that there exists a strict inequality $s \neq 0^m$ in $\mathcal{B}^i_{<}$ with variable $(\mu^i)_s$ such that $(\mu^i)_s < 0$. Then this will lead to a strict inequality in the transformation from (37) to (38), since the variable

is not zero.
Now assume for i that $(\beta^i)^T \mu^i - d_0 > 0$, i.e. the inequality (25) is strict. Then in the transformation from (38) to (39) the inequality would be strict.
So in either case the points on the border are not part of any B^i and therefore $d^T x \leqslant d_0$ separates the regions. □

Alternatively, a linear program that is forced to use a strict inequality of the right-hand side of the linear combination evaluates if d_0 can be used.
We want to point out that the condition (23) is crucial for these cases. Otherwise the solution $(d, d_0, \alpha, \beta) = 0$ is always a feasible solution, and therefore other meaningful solutions with the same optimal value will possible be ignored.

5.1.3.2 *Greedy Approach*

The LP (21-28) is trying to separate all regions by always forcing A^i on one side of l^* and all B^i on the other side of l^*. This is more than we actually need in order to satisfy Proposition 22. So we expand our LP-problem by a greedy approach as follows. Assume we find a linear constraint separating the pairs (A^i, B^i) for the first $k-1$ convex pairs, but the LP (21-28) does not find a solution when trying to set A^k on the one side and B^k on the other. Then we try to switch the sides of A^k and B^k, i.e. we modify the variable bounds and the constraints concerning the last added pair of regions, such that the convex regions will change the sides on l^*.
The LP (21-28), Lemma 23, and Proposition 24 can be adopted easily for this approach.
This is only a greedy approach, as we only change the positions of the convex regions of the last iteration.
In [2] they face a similar problem. There for every LP they will solve an SMT-problem without running into the problem to solve this last step with an greedy approach. They introduce Boolean variables, for the decision on which side the pair will be.

5.2 OPTIMIZATIONS

Our main goal with the optimizations is to reduce the number of tested pairs of linear constraints to a reasonable amount. For this goal, we first introduce the concept of NRC points for interpolant constraints, which are then used to choose interesting candidates of pairs of linear constraints. The idea behind this heuristic is that it is more likely to combine constraints when they are needed to separate the same regions. There will be situations where we will not choose the correct pair. To check the potency of the heuristic choice of specific pairs and the overall heuristic algorithm we computed all

benchmarks in Section 8.3 with the heuristic choice and by testing every pair of interpolant constraints. Furthermore, we give some other optimizations to the general approach.

5.2.1 NRC-*Points*

An NRC-point (a, b) is a pair of points, where $a \in A$ and $b \in B$ are only distinguishable by one interpolant constraint h and are indistinguishable for every other interpolant constraint, we call (a, b) an NRC-point of h. Formally, an NRC-point (a, b) of h is a solution of the formula

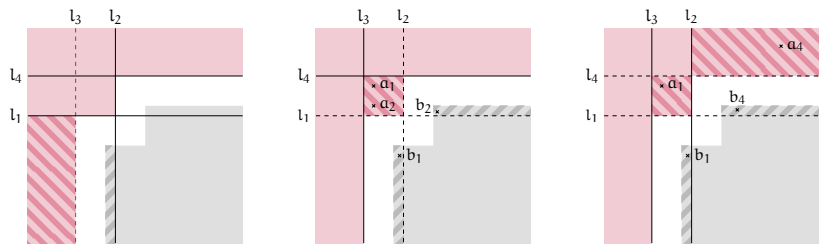
$$(a \in A) \wedge (b \in B) \wedge (a_{\mathbb{B}}B = b_{\mathbb{B}}B) \wedge \left(\bigwedge_{l \in L \setminus h} l(a) = l(b) \right) . \quad (40)$$

Since there can be more than one NRC-point of h, we first solve (40), then we compute convex regions C_a, C_b around a and b, with the method described in Section 5.1.2. After this, we can solve (40) with the additional conjunction that $(a \notin C_a) \wedge (b \notin C_b)$. This can be done multiple times, and with the advantage of an incremental SMT since we only add clauses to the formula.
It is worth mentioning that the search for the NRC points detects if an interpolant constraint is redundant, i.e. it is not needed to fulfill Proposition 22. This is the case, when (40) is not satisfiable in the first iteration. When this occurs, we delete the inequality from the interpolant constraints. To keep the computational effort low, we only compute a maximum of three NRC points for each interpolant constraint, this is motivated by experimental results.

Example 5 (cont.)**.**
Consider Example 4, we want to compute all NRC *points for* l_3*. Therefore we search for points* (a_3, b_3) *that are a solution to the problem* $(a_3 \in A) \wedge (b_3 \in B) \wedge (\bigwedge_{i \in \{1,2,4\}} l_i(a_3) = l_i(b_3))$*. The pair* (a_3, b_3) *is shown in Figure 8* (a)*. The highlighted areas are again the convex regions built around the points* a_3, b_3*. There is no other solution that is not in the convex regions, therefore* (a_3, b_3) *is the only* NRC*-point for* l_3*. Additionally it is easy to see in Figure 4 that the pair* (a_1, b_1) *is an* NRC*-point of* l_1*, and* (a_2, b_2) *an* NRC*-point of* l_2*.*

5.2.2 *The Choice of Interesting Pairs of Inequalities.*

After we calculated the NRC points for every interpolant constraint, we compare the constructed convex regions around these points. A pair of inequalities (s, t) is chosen by our heuristic, if there exist two NRC points (a_s, b_s) and (a_t, b_t) for interpolant constraints s and t, such that either a_s and a_t or b_s and b_t are equal in respect to



(a) (b) (c)

Figure 8: Example of NRC points and the heuristic choice of pairs.

all $l \in L \setminus \{s, t\}$. In this case, our heuristic chooses the pair (s, t) as a pair that is promising, and therefore will be tested by the method described in Section 5.1.
If it was possible to improve the interpolant by testing the pair (s, t), all other constraints that were chosen to be tested with either s or t are then tested with the newly found interpolant constraint.

Example 6 (cont.)**.**
Consider Example 5, we already computed NRC *points for* l_1, l_2 *and* l_3*. The heuristic for interesting pairs now compares these points, i.e. the convex regions around these points only built by interpolant constraints. In Figure 8* (b)*, we see that the convex regions around* a_1 *and* a_2 *are the same, the heuristic therefore chooses the pair* l_1, l_2 *as a promising candidate. The heuristic does not choose the pair* l_1, l_4 *since none of the convex regions are equal as we can see in Figure 8* (c)*.*

5.2.3 *Using NRC-Points to Save* SMT *calls.*

The time consumption for solving a new SMT is huge, therefore we try to minimize the amount of SMT-problems we have to solve. One way is to use NRC points to skip the computation of unsatisfiable points, i.e. solving the formula (18) initially. Whenever we want to replace two interpolant constraints l_1 and l_2 by a new linear constraint l^*, the algorithm would first compute a pair of points that is not separable. Since l^* is typically initialized by one of the constraints, e.g. $l^* = l_1$, we can use the NRC points of l_2 as the first indistinguishable points.
If we are not in the initial search of indistinguishable points we will still search through all NRC points of both linear constraints and check whether or not these points are still separated.
Since we will use NRC points quite frequently in this way it is worth updating them in the process of the algorithm. Therefore when we solve an SMT-formula by the solver the found pair is added as an NRC-point to both linear constraints l_1 and l_2. Additionally when

we successfully merged them to a single linear constraint l^* this constraint will inherit all NRC points of l_1 and l_2.

5.2.4 *Simplifying the Description of the Convex Regions.*

There are benchmarks where we detected a lot of constraints having the same normal. Inspired by these cases we want to avoid the introduction of unnecessary variables in our linear programs with a fast redundancy test. Since all constraints have to be orientated such that the inequality-sign is $<$ or $\leqslant$, we go through all constraints that are computed by our method described in Section 5.1.2 and take the lowest right-hand side if there are multiple constraints with the same normal. If the normal and the right-hand side of two constraints are the same, we take the strict inequality in preference to the non-strict inequality.

5.2.5 *Use Rational LP-Solvers Only When Needed.*

Most of the time we only need the LP to verify that the current pair of constraints cannot be substituted by one new constraint. Therefore we will first use an LP solver without rationals to solve the problems. If it states that the LP has no valid solution we will skip the pair, if we find a solution we will make a rational constraint and check whether the previous pair of points is separated by the new constraint. Further we will test in the next iteration if we encounter the same convex region as in the previous iteration. If it won't separate the pair of points, or the same convex region appears in the next iteration we will change to a rational LP solver for the rest of the computation if one constraint is sufficient for the current pair. For the next pair of constraints we will again begin with the LP solver without using rationals, and only change to it if we need to.

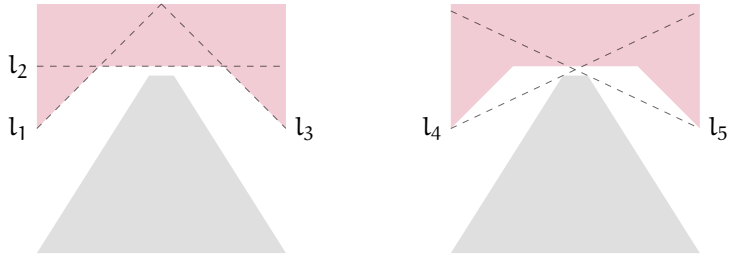
6 LOWER BOUNDS FOR TOUCHING STATE SETS

In this chapter we want to compute lower bounds for the *Minimize Interpolant Problem*, stated in Definition 15. We will only give a few statements for the general case and then will focus on the case where the state sets will touch each other. The main purpose of this computation is to identify already optimal benchmarks, and to get a measurement for the quality of the computed interpolants. In many cases though the lower bound will be far off the actual global minimum, because we only have reasonable techniques for touching state sets. Even for them we will only find the global minimum if there are at most two non-touching constraints.

6.1 THE GENERAL CASE

In the general case we are only able to state trivial lower bounds. Since we are talking about an amount of linear constraints, the trivial bound is 0. When it is the case that no linear constraint is needed to separate the state sets, our algorithm will recognize it, since the test for redundancy will state that every linear constraint is redundant. In the case where actual linear constraints are needed to separate the state sets we can only give 1 as a lower bound, regardless of how many constraints are in the computed interpolant.
There are in fact situations where it is possible to give better lower bounds. For example if we would recognize that the same convex region is used multiple times in one LP. We can reduce the LP, by using only pairs of convex regions that have the region as a component. If such an LP is not solvable, then we can definitely say that the two linear constraints are not replaceable by one new constraint, and therefore we can increase the lower bound to two. But if this happens multiple times we are not able to increase the lower bound further than two. The following example will show a situation where this happens in two different situations, but the needed interpolant contains still only two linear constraints.

Example 7.
In the example shown in Figure 9 (a) both pairs (l_1, l_2) *and* (l_1, l_3) *have the situation that their* LP *can be written in the reduced form with all pairs included, and still they are not separable.*
But as we can see in Figure 9 (b) the linear constraint l_4 *and* l_5 *are sufficient to build an interpolant.*



(a) (b)

Figure 9: Example for the Worst Case for Lower Bound

6.2 TOUCHING STATE SETS

During the model-checking algorithm our method can be called to refine a previously computed interpolant. In this case the second state set is not created by epsilon bloating, but is the inverse of the previously computed interpolant united with the spurious counterexample that was found. This is done to have a monotony on the computed interpolants, i.e. the new interpolant will always be a subset of the old interpolant. But since we try to minimize the overapproximation and therefore, if possible, let the interpolant constraints touch the initial state set, it will often occur that we actually have used a constraint in the interpolant that is used in the description of the state set itself. In the following theorem we will use the terminology of simple topology, i.e. for a set A we will use $\bar{A}$ for its closure, $\delta(A)$ for its boundary. Since our sets are in the space $\mathbb{B}^n \times \mathbb{R}^m$ this is simply induced by the euclidean space, where the different real spaces are separately handled by comparisons of the Boolean part.

Theorem 25.
Let $A, B \subset \mathbb{B}^n \times \mathbb{R}^m$ be two disjoint state sets. If there exists a hyperplane h, such that

$$h' = h_= \cap \delta(A) \cap \delta(B) \tag{41}$$

includes a $(m-1)$-dimensional convex polytope C, then every finite set of linear constraints that suffices the condition of Proposition 22 for these state sets contain a linear constraint which corresponds to h, i.e. all points that suffice the linear constraint with equality are in $h_=$.

Proof. Assume that h' includes an $(m-1)$-dimensional convex polytope C. Furthermore, let H be a set of hyperplanes that suffices the condition of proposition 22, but none of the hyperplanes corresponds to h, i.e. for every hyperplane $\widetilde{h} \in H$ it holds that $\widetilde{h}_= \cap h_= \neq h_=$.
Then for every point in C we can create a pair of points in $A \times B$ with

distance ε. For each of these pairs, and for every ε there must exist a hyperplane in H, but since none of them corresponds to h, each of them can only cover an $m-2$-dimensional subspace of C. To separate C we therefore need an infinite amount of hyperplanes which contradicts the condition that H is a finite set.

□

We can therefore use this theorem to raise the lower bound of hyperplanes for touching state sets. Therefore we have to compute the closure of both sets. Then we choose one state set, e.g. the state set with the smaller number of hyperplanes. For every hyperplane h in the description we will compute $h' = h_{=} \cap \delta(A) \cap \delta(B)$, and the dimension of h'. If the dimension of h' is $m-1$, we have to check if it includes a $(m-1)$-dimensional convex polytope. When all tests were positive we have to check if another hyperplane already corresponded with h. The last step is necessary, since the description could be using a redundant description, and even if it is not redundant, there could be a lesser and a lesser-or-equal hyperplane that is used in the description in different parts of the state set.
There are cases where we need two hyperplanes that correspond to the same h, in this case they describe a lesser and a lesser-or-equal hyperplane in the same orientation. This is only needed if there are different $(m-1)$-dimensional convex polytopes that are already part of A and B, i.e. there exists a $C_A \in A$ and $C_B \in B$ with $C_A, C_B \subset h'$. Finally we have a set of constraints H.
The final lower bound for touching state sets is computed by the number of constraints in H, and the fact that was stated in the general case. If the constraints found due to the theorem are not enough to separate the state sets, we can conclude that the lower bound is at least $|H| + 1$.

6.3 IMPLEMENTATION

We only implemented the case of touching state sets, since the bounds in the general case are to small to being relevant. In the theoretical view we talked about the closure and proved Theorem 25 such that we will have these closures for our state sets. Unfortunately that is not the case with the description of state sets given as an LINAIG. It is not obvious how to compute a closure of a state set with an SMT-formula. We did it in a naive way, and had to make up for it, because what we compute is not the actual closure of the set, but we will take care of it in the algorithm afterwards.
The naive way we implemented the closure for a state set A is the following. We copied the complete structure of A into a new predicate A', and then replaced every literal that was a linear constraint by almost the same linear constraint, except for its sign. This was switched so that the equality of the constraint changed the side, e.g. $a^t x \leqslant b$

is replaced by $a^T x < b$. The naive closure is then $A \vee A'$. Note that the whole border of A is described by linear constraints, therefore the only parts that are in the closure but not in A are describable by linear constraints, more explicit by an SMT-formula with linear constraints. By changing the sign from lesser-than-equal to lesser-than we will include the facets that are created by this constraint either in A or in A'. We abuse here the word facet, this is a known term in geometry of polytopes and describes an $(m-1)$-dimensional face of an m-dimensional convex polytope. It is appropriate to use it in this context because a state set could be described by a union of polytopes and therefore its border is always a union of facets of convex polytopes.

Further we are only interested in parts of the closure that can include an $(m-1)$-dimensional convex polytope. Therefore it is okay that $A \vee A'$ may miss lower dimensional faces, we again abuse the term face that is known from geometry of convex polytopes, e.g. edges or vertices.

The last and most crucial point is that we might include points that are not part of the closure. This can happen when a linear constraint is present in A with both orientation, i.e. $a^T x \leqslant b$ and $a^T x \geqslant b$. They can be used to describe the empty set by asserting $\neg\ (a^T x \leqslant b)$ and $\neg\ (a^T x \geqslant b)$. In the naive closure we will therefore also include $\neg\ (a^T x < b)$ and $\neg\ (a^T x > b)$, which will include the equality. This fact is also a known problem in the epsilon bloating. All cases are shown in Figure 10.

We therefore start the algorithm by computing the two naive closures of the disjoint state sets A and B. Then we construct three different SMT-instances. The first two are just initialized with $not(A)$ or $not(B)$, respectively. The last instance is initialized with the naive closures A and B. Now we solve the last SMT-problem, and if it is unsatisfiable, i.e. no point is in the intersection of the naive closures of A and B, the state sets A and B are not touching state sets. In the case it is satisfiable we get an assignment for the variables, and we have to ensure that it is a point of the real closure. Therefore we do the following.

We have to check this for A or B, respectively. Here we only describe the computation for A, the case of B is analog. First we create a convex region around the real point by the assignment, similar to Section 5.1.2 but with the difference that we will ignore linear constraints that are satisfied with equality. Then we assert this convex region and the Boolean assignment to an SMT-problem that was initialized with $not(A)$. If this is satisfiable it means that there are points around the assignment, i.e. in the convex region that are not in A, which is in fact what we are looking for. But if it is unsatisfiable it means that the whole convex region is in A, therefore the assignment cannot be part of the real closure of B, since A and B are disjoint.

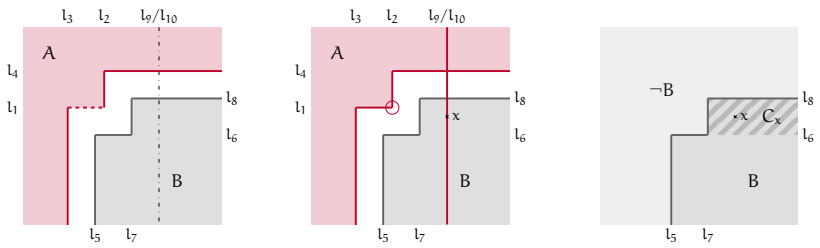
If both tests are satisfiable we will continue by collecting the set of lin-

ear constraints of the naive closures of A and B that are satisfied with equality. For this test every constraint must be normalized. Again we construct a convex region around the point, in this case considering every constraint that is used in the description of the naive closures of A and B. Then we test how many of the constraints in the set satisfy the whole convex region with equality. If this is only one constraint this convex region is $m-1$ dimensional and therefore suffices the condition of Theorem 25. Afterwards we look up what constraints are used in A and B that correspond to the equality from the found constraint. In the case that one state set uses only one constraint we can use this as a certain interpolant constraint, but if both state sets use both possible descriptions we know that we need at least one of these constraints, but not certainly if the other one is mandatory as well.
The last step of an iteration is to exclude the assignment that was found. In the case that we found a constraint that has to be present in the interpolant we exclude not only the convex region but the whole equality of that constraint, since it is already proven that the constraint is needed, and by this we will not lower the dimension of any other convex region with dimension $m-1$. Note that we will probably remove vertices by this procedure, which is a good thing, since they do not contribute to a lower bound.

Example 8.
In the example shown in Figure 10 (a) the state set A of Example 2 is slightly modified. The constraint l_1 is a lesser-then constraint, and therefore the dashed line is not part of the state set. Additionally we introduced l_9, l_{10}, and they only differ in the sign, i.e. if l_9 is described by $a^T x \leqslant b$ then l_{10} is $a^T \geqslant b$. The state set A is described by the following formula

$$A = (l_1 \wedge l_2) \vee l_3 \vee l_4 \vee (\mathrm{not}(l_9) \wedge \mathrm{not}(l_{10}))$$

The dashed-dotted black line indicates presence of the linear constraints, but nothing is added to the state set by expanding the formula with $\mathrm{not}(l_9) \wedge \mathrm{not}(l_{10})$.
In Figure 10 (b) we show the result of the naive closure. The previously



(a) (b) (c)

Figure 10: Example of the naive closure and its differences to the real closure

dashed-dotted line is now actually a part of A'*, because there* l_9 *and* l_{10} *are changed to* $a^T x < b$ *and* $a^T > b$*, and therefore* $\mathtt{not}(l'_9) \wedge \mathtt{not}(l'_{10})$ *includes the line segment. The circle around the vertex of* l_1 *and* l_2 *indicates that it is not present in the naive closure, since it is neither in* A *nor in* A'*. The point* x *indicates a satisfying assignment of the intersection of both naive closures.*

Finally in (c) we see an example of the check whether x *is in the real closure of* A*. The convex region* C_x *has no intersection with* $\mathtt{not}(B)$ *and therefore* x *cannot be part of the real closure of* A*.*

7

APPLICATION: DESCRIPTION TEST

Since model-checking is used in the verification of safety properties, every computation is made in rational arithmetic. This leads to situations, where through the iterations the description of the actual state set, i.e. the linear constraints, are using fractions with large numerators and denominators as factors, which are time consuming in doing further computations in exact arithmetic. This fact leads to the application of our method for improving the description of a given state. The idea that this has to be done is not new, i.e. a well known tool for verification of hybrid systems PHAVer [27] described how they handle bad coefficients in linear constraints. Their case differs in multiple ways to ours, but we used it as a basic algorithm to start with. First, they consider only convex polytopes, where we have to handle the whole state set as ones, which is a union of convex polytopes. Second, the resulting overapproximation is in their case not limited by any other structure, where in our case we still want to ensure a reasonable overapproximation, i.e. there is a second state set which has to be a superset of the computed overapproximation, see Section 2.4.5. Despite these facts we are still able to use major thoughts of their idea.
We will first describe their algorithmic idea and then modify it in the next section, such that it fits to our circumstances. Their goal is to find an overapproximation of a convex set P, such that the used constraint $a_i^T x \leqslant b_i$ is replaced by $\alpha_i^T x \leqslant \beta_i$, while the coefficients α_{ij}, β_i each uses less than z bits. The approach of PHAVer is, to scale down a_{ij} so that these coefficients are smaller than z bits, and then find a β_i that will overapproximate the polytope. Their idea is to calculate a scaling factor f, such that the new coefficients can be represented as

$$\alpha_{ij} = f a_{ij} + r_{ij}$$
$$\beta_i = f b_i + r_i$$

where r_{ij} and r_i are rounding errors. Since they round the coefficients for the normal to the next integer, the error $|r_{ij}| \leqslant 0.5$ is bounded. The rounding error of r_i is based to the new normal α_i, and therefore not bounded. To find a good representation the goal is to find a scaling factor f close to 1. To get an initial assumption for f they choose for β_i the closest integer, which will also result in a bound for $|r_i| \leqslant 1$. This leads to upper bounds for the scaling factor f:

$$f \leqslant (2^z - 3/2) / |a_{ij}| \text{ and} \tag{42}$$
$$f \leqslant (2^z - 2) / |b_i| . \tag{43}$$

Algorithm 2 : Limiting the number of bits in a constraint [27]

input : Polyhedron as a set of constraints
$P = \{a_k^T x \sim_k b_k | k = 1, \ldots, m;\ \sim_k \in \{\leqslant, <, >, \geqslant\}\}$,
index i to constraint to be limited,
desired number of bits z

output : new constraint $\alpha_i^T x \sim_i \beta_i$

begin

$success := false$
$f := min\{(2^z - 3/2)/|a_{ij}|, (2^z - 2)/|b_i| \mid j = 1, \ldots, n\}$
while $\neg\, success$ **do**
  for $j = 1, \ldots, n$ **do**
    $\alpha_{ij} := round(f a_{ij})$
  $q := inf_{x \in P} \alpha_i^T x$
  if $\alpha_i = 0 \vee q = -\infty$ **then**
    return success
  $\beta_i := ceil(q)$
  if $|\beta_i| \leqslant 2^z - 1$ **then**
    $success := true$
  else
    $f := min\{f/2 - 3/(4|a_{ij}|), (2^z - 2)/|\beta_i| \mid j = 1, \ldots, n\}$
return success

Then they employ a heuristic to solve this, shown in Algorithm 2. First they estimate the scaling factor with (42) and (43), then they calculate a factor q by linear programming and set $\beta_i = ceil(q)$. If β_i has less than z bits, they found a solution, otherwise they repeat the process by decreasing f and starting over. The procedure terminates if β_i has less than z bits, or the factors α_{ij} are all set to zero, in which case the problem is infeasible, or if the LP is not solvable.
We want to state out some things that are formulated a bit sloppy or where we disagree in their choice.
Mainly that is the case in the choice and computation of f. The goal is to limit the number of bits that every single coefficient is using, i.e. $|\alpha_i j| \leqslant 2^z - 1$ holds for every coefficient and for the right-hand-side. This way it is ensured that α_{ij} can be stored by using z bits with the two's complement. They state in their paper that in further iterations they want to decrease f, but in the pseudocode they do not handle the case that factors a_{ij} may be zero. This would lead to the fact that f is no longer a scaling factor between 1 and 0, but rather can be $-\infty$ or ∞. Since the goal is to limit the number of bits, for those coefficients there is absolutely no need to lower f in any kind. We know that this fact is pretty obvious and is just space-consuming in a pseudocode, but we still think this should have been mentioned. The other fact is that the behavior of f is quite different from the first iteration to all following iterations. In the first iteration the coefficient

with the largest absolute value will define f, which makes sense, because in this way every coefficient is forced to be smaller than the limit $2^z - 1$. But in all following iterations the computation of f is lead by the smallest absolute values and the crucial fact is that they can lead to negative scaling factors, which is not intended. Our guess is that in each iteration they want to reduce the coefficients to use one bit less, such that b can be one bit less. It is like scaling the whole linear constraint by $1/2$ but with the fact that they have to recompute the right-hand-side, because they round every coefficient. This guess can be formalized as

$$\left|\alpha_{ij}^{(s)}\right| \leqslant 2^{z-s} - 1 \,, \tag{44}$$

when the first iteration starts with $s = 0$. We will sketch how this mistake might have happened. If one wants to prove that $f^{(s+1)} := \min\{f^{(s)}/2 - 3/(4|a_{ij}|), (2^z - 2)/|\beta_i| \mid j = 1, \ldots, n\}$ is the right formula, one would make a proof by induction, there it would be fine to use $f^{(s+1)} \leqslant f^{(s)}/2 - 3/(4|a_{ij}|)$, since we took the minimum of all that factors. Let us look at the proof. The inductive basis is valid. Let (44) be valid for all previous s, and prove that it is valid for $s+1$,

$$\begin{aligned}
|\alpha_{ij}^{(s+1)}| &= \left|a_{ij} f^{(s+1)} + r_{ij}\right| \\
&\leqslant \left|a_{ij} f^{(s+1)}\right| + |r_{ij}| \\
&\leqslant \left|a_{ij}\right| \left|f^{(s+1)}\right| + 1/2 \\
&\leqslant \left|a_{ij}\right| \left|\frac{f^{(s)}}{2} - \frac{3}{4|a_{ij}|}\right| + \frac{1}{2} \\
&= \left|\frac{f^{(s)}|a_{ij}|}{2} - \frac{3}{4}\right| + \frac{1}{2} \,.
\end{aligned}$$

If we ignored the absolute value, we would conclude the formula that was given in the paper. To disprove the statement that (44) is fulfilled in every iteration we simply need to look at an example.

Example 9.
Consider two coefficients $a_0 = 60$ and $a_1 = 1$, and the limit of bits defined by $z = 5$. We compute the first iteration $f^{(0)} = (32 - 3/2)/60$. Now consider that the right-hand-side is still not good enough, and we compute the second iteration.

$$f^{(1)} = \min\left\{\frac{32 - 3/2}{120} - \frac{3}{240}\,,\ \frac{32 - 3/2}{120} - \frac{3}{4}\right\} = \frac{32 - 3/2}{120} - \frac{3}{4} = -\frac{119}{240}$$

So this time we look at the new coefficient

$$\begin{aligned}
\mathrm{alpha}_0^{(1)} &= \mathrm{round}\left(f^{(1)} a_0\right) \\
&= \mathrm{round}\left(-\frac{119}{240}\, 60\right) \\
&= -30 \,.
\end{aligned}$$

Therefore $|\mathtt{alpha}_0^{(1)}|$ *is not lesser or equal* 15.

The main problem why we mention this mistake is that the algorithm will not terminate, since it does not hold that $\lim(f) = 0$ for $s \to \infty$, and therefore the termination criterion that $\alpha_i = 0$ will never trigger. To modify the algorithm there are multiple options. We choose to modify the computation of f to only consider the largest absolute coefficient. By doing this, the iteration steps are limited by $z + 1$, where in the last iteration it will always end up with $\alpha_i = 0$.

7.1 MODIFIED ALGORITHM FOR BETTER STATE SETS

We have to modify the algorithm to handle our cases. There we do not handle polytopes but state sets, and further we are searching for a better representation in a restricted area, i.e. we again compute an interpolant of two state sets.
First the linear program in Algorithm 2 only computes a q, such that the linear constraint described by α_i is tangent to their polytope. In our case the linear constraint is used in the description of a state set, and therefore possibly used in the description of multiple polytopes. This problem is handled by making use of our algorithm concept of interpolants. We will not use the complete description of the state set but rather look for significant pairs of convex regions. First we search for NRC points of the constraint, which representation is currently optimized. These points are extended to convex regions C_s, C_t. In the first iteration, where only one pair of convex regions is present, this will be similar to the algorithm showed in the previous section. But since we need to handle multiple convex regions at once, we modify the LP $q = \inf_{x \in P} \alpha_i^T x$ to a formulation of two LPS that are somehow separations of the LP described in 5.1.3 with a fixed normal d, another objective function, and without the use of an extra variable ε that defines the margin.

$$\begin{aligned} \min \quad & d_0^{(1)} \\ \text{s.t.} \quad & (S^k)^T \lambda^k = d \\ & (s_0^k)^T \lambda^k \leqslant d_0^{(1)} \\ & d = \alpha_{ij} \\ & \lambda^k \geqslant 0 \end{aligned} \tag{45}$$

$$\begin{aligned} \max \quad & d_0^{(2)} \\ \text{s.t.} \quad & (T^k)^T \mu^k = d \\ & (t_0^k)^T \mu^k \geqslant d_0^{(2)} \\ & d = \alpha_{ij} \\ & \mu^k \leqslant 0 \end{aligned} \tag{46}$$

The LP (45) can be viewed as a direct extension of the one used in the previous algorithm, where as (46) is used to ensure that the new constraint is in fact separating these convex regions. Similar to our case in Section 5.1.3 we have to decide which convex region has to lay

on which side of the constraint, but differently in this case we already know the original constraint, and therefore the convex region C_s is added to (45) if the point s satisfies the original constraint, or to (46) if it is not.
As solutions for these LPs we get two right-hand-side values $d_0^{(1)}, d_0^{(2)}$ for the normal d. From the first value we compute $\beta_i = ceil(d_0^{(1)})$, and if $\beta_i < d_0^{(2)}$ the new constraint $\alpha_{ij}^T x \leqslant \beta_i$ is feasible for our problem, i.e. separates the convex regions that were introduced to the LPs.

Lemma 26.
If both LPs *(45,46) are solvable, and* $ceil\left(d_0^{(1)}\right) < d_0^{(2)}$ *then the new constraint* $\alpha_{ij}^T x \leqslant \beta_i$ *separates every* S^k *from* T^k, *i.e.* $\forall k \forall x \in S^k \alpha_{ij}^T x \leqslant \beta_i$ *and* $\forall k \forall x \in T^k \alpha_{ij}^T x > \beta_i$.

The proof of this lemma is analogous to the one given for Lemma 23. So after this we check if the quality is good enough, and if not we have to compute another iteration.
In our case we will not terminate the whole algorithm if we found a new constraint, which has the right size of coefficients. Instead as in our original algorithm we have to make sure that everything is separable, i.e. the interpolant constraint where the i-th constraint is substituted by the new constraint fulfills the condition of Proposition 22. This again is checked by the same procedure we described in Section 5.1.1, so if a new pair of indistinguishable points is found by the SMT we will start a new iteration. The LPs will therefore grow with each of these iteration by one convex region.
Further we were not pleased with the results. When the original description of the constraint needs more than 150 bits to represent a coefficient, it is not enough to stop if there is no description with 16 bits, because also a 32 bit representation would be a good improvement. On the other it is preferable to use integers instead of fractions if possible, e.g. 1 instead of 10000000000/10000000001. To achieve this we have to do a bit more. We can start with the lowest z such that 2^z is lower than the worst coefficient. Then we save the first iteration where all coefficients, including the right-hand-side, are lower than 2^z. Even further we will try to improve the description even further by simply reducing the coefficients with the same technique as before. If this tends to be solvable, we will save the improved version and continue reducing, if it fails we will use the last saved constraint. This constraint is then used as we described above, we have to check if everything is separable.
Alternatively we can increase the number of bits, when the algorithm is failing. The modified algorithm is shown in Algorithm 3.

Algorithm 3 : Modified version to limit the number of bits in a constraint of a state set

input : State set A with a set of constraints
$I = \{a_k^T x \sim_k b_k | k = 1, \ldots, m;\ \sim_k \in \{\leqslant, <, >, \geqslant\}\}$,
State set that bounds the overapproximation B,
index i of the constraint to be limited,
maximum number of bits max_z

output : new constraint $\alpha_i^T x \sim_i \beta_i$

begin

- $z = 1$
- $I := I \setminus a_i^T x \sim_i b_i$
- $(\alpha_i, \beta_i) = 0^{n+1}$
- $I := I \cup \{\alpha_i^T x \leqslant \beta_i\}$
- $max_a = \max\{|a_{ij}| \mid j = 1, \ldots, n\}$
- **while** *true* **do**
  - $success := false$
  - **if** *there is a none separated pair of Points* s, t **then**
    - **if** $a_i^T p \sim_i b_i$ **then**
      - Add C_s to the first LP (45)
      - Add C_t to the second LP (46)
    - **else**
      - Add C_s to the second LP (46)
      - Add C_t to the first LP (45)
    - $success := false$
    - $f := \min\{(2^z - 3/2)/max_a, (2^z - 2)/|b_i|\}$
    - $restart = false$
    - **do**
      - **do**
        - **for** $j = 1, \ldots, n$ **do**
          - $\alpha_{ij} := round(f a_{ij})$
        - Solve the linear programs with fixed normal $d = \alpha_i$ and get $(d_0^{(1)}, d_0^{(2)})$
        - **if** $(\alpha_i = 0) \vee$ (*one lp is not solvable*) **then**
          - $z := 2z$
          - **if** $z < max_z$ **then**
            - $restart := true$
          - **else**
            - **return** false
        - $\beta_i := ceil(d_0^{(1)})$
        - **if** $(|\beta_i| \leqslant 2^z - 1) \wedge (\beta_i < d_0^{(2)})$ **then**
          - $success := true$
        - **else**
          - $f := \min\{f/2 - 3/(4max_a), (2^z - 2)/|\beta_i|\}$
      - **while** $\neg\, restart$
    - **while** $\neg\, success \wedge \neg\, restart$
  - **else**
    - **return** $(true, \alpha_i^T x \leqslant \beta_i)$

8

EXPERIMENTS AND EVALUATION

This chapter presents the experiments of our algorithms. First we describe the experimental setup in Section 8.1. Then we evaluate the algorithm for the lower bound computation of touching state sets in Section 8.2. The results are used to determine a more appropriate subset of the test instances for evaluating the interpolant methods. Further we evaluate the different optimization strategies in Section 8.3, and choose two parameter settings. In Section 8.4 we compare the two chosen parameter settings of our algorithm to the other approaches mentioned in Section 4. Finally we evaluate the description test on the same set of instances in Section 8.5.

8.1 EXPERIMENTAL SETUP

For benchmark instances we extracted intermediate state sets that occur during runs of the model-checking algorithm for a variety of models. The models, and therefore the instances we extracted, differ in the amount of Boolean and real variables and also in the complexity of the interpolants. Since for evaluating the interpolant methods it is not important how well the model checker itself performs, we increased the times where the model-checker algorithm would use an interpolant method to get a reasonable amount of instances. We can differentiate them by whether they occurred during a normal approximation or during the computation of a refinement step.
Further we added a time limit for the computation of our methods $A_1, \ldots, A_{17}$ to 600 s, but if there was a successful reduction during the computation we compute the new interpolant with the current interpolant constraints. Otherwise we output the interpolant that was given as an input to our algorithm. Since we use the interpolant computed by the constraint minimization as an input, the runtime of our algorithms always includes the computation of the interpolant.
We will not take instances into account where the constraint minimization, Section 4.1, exceeds 200 s of time limit, and instances in which no constraint is needed to construct an interpolant. For the evaluation we divided the models in different categories: *acc*, *flap*, *intersect*, *dam*, and *etc*. In the *acc* models the objective of the controller is to set the acceleration of the controlled car to reach the goal velocity in a distance equal to or greater than the goal distance.
Flap contains models where a controller corrects the pilot's commands if they endanger the flaps, which are used during take-off and landing to generate more lift due to high velocity.

In *intersect* are models that monitor the traffic at an intersection. The controller has to guarantee a car on the minor road can safely drive over the major road without interfering its traffic.
Dam consists of models, where a controller of a water dam has to switch turbines on and off. After a certain amount of switching operations the turbines need to enter a maintenance mode. The controller has to guarantee that the water level always stays within given bounds.
In *etc* the models are derived from a collision avoidance protocol for trains, consisting of speed supervision and a cooperation protocol to a radio block center that grants movement authorities to the train. For more details on the models we refer to [4].
We also have a second set of instances without further information of the refinement step, the overlying model, and the epsilon value. These instances were also created with the same model-checking algorithm. In the evaluation we will separate those two sets of instances, since we can extract some information out of the more detailed context of the first set. Since the overall amount of instances in the first set is quite small, we take a second view on all instances together, where we can extract more general information.

8.2 EVALUATION FOR LOWER BOUNDS

The objective for this evaluation is to prove that there are cases, where we can verify the optimality of the given interpolant. Additionally, we want to create a subset of benchmarks for the evaluation of the interpolation methods, where such cases are excluded.
In the first test set we focus on the cases of refinement, due to the fact that we did not propose any real lower bound computation of the general case. Further we can evaluate if a given interpolant is already globally optimal, i.e. if the lower bound is equal to the computed number of linear constraints.
Therefore we computed for every instance that arose from a refinement step the lower bound as described in Section 6. Table 2 differentiates between model category and epsilon value. The columns state the runtime (time) used to compute the lower bound, the total amount of instances (# instances) and the number of instances where the initial interpolant of the constraint minimization is already optimal (# cm opt), as well as the average size of the lower bound (lb size) and the constraint minimization (cm size).
There is a significant difference between the model categories and the epsilon values. When the epsilon value is set to infinity, more refinement instances arise. This is not surprising, since the first overapproximation is much more likely to create a spurious counterexample, than in the case of 0.05. The main amount of instances in total is based on the *avc* category, where 2073 instances arose with an epsilon

Table 2: Results for the Lower Bound Computation for Refinement Instances of the First Test Set

	type	#inst.	time [s]	lb size	cm size	#cm opt.
$\varepsilon = \infty$						
	avc	2073	4.04	12.75	13.35	1894
	dam	17	26.31	1.00	34.53	0
	etc	9	1.48	1.56	2.78	3
	flap	32	2.56	2.81	3.75	21
	intersect	204	4.29	4.05	6.80	33
$\varepsilon = 0.05$						
	avc	35	107.96	6.06	13.17	2
	flap	12	147.85	15.00	26.00	0
	intersect	103	187.80	16.66	24.81	9

value set to infinity. In this setting also the most instances (1894) are already optimal due to the constraint minimization, this means that over 91 % of these instances are not suitable for testing an interpolant method.

The results of Table 2 show that over 78 % of the instances of the refinement are already restricted. We observe that for the few instances where the epsilon is 0.05 %, the lower bound computation seems much harder. Also we observe that the refinement seems to be model-specific, e.g. the lower bound for all *dam* models is trivial, but not easy due to the computation time needed. All cases where we confirmed that the initial interpolant is already optimal will be ignored in the following evaluations of the interpolant methods.

Since the second test set has no information of its occurrence in the model-checking process we computed the lower bound on every instance. This means that all instances that occurred not in a refinement step will most likely not have touching state sets and therefore lead to trivial bounds. Table 3 uses the same rows as Table 2, only without further differentiation. The row *unknown* consists of all instances of the second test set, while the second row *all* is the combined test of the first and the second test set. In the evaluation of the second test set we can state that only about 6 % of the instances are already restricted. Recall that we do not know if the instances arose from refinement steps. After this evaluation of the lower bounds we end up with 674 instances in the first test set, and 6324 in the second test set, for further evaluations.

Table 3: Results for the Lower Bound Computation of the Second and the Combined Test Set

	#inst.	time [s]	lb size	cm size	#cm opt.
unknown	6757	0.41	1.94	7.21	432
all	9242	4.06	4.61	8.86	2394

8.3 EVALUATION OF OPTIMIZING STRATEGIES

We will first take a look at the performance of the optimizing strategies stated in Section 5.2. Table 4 shows all algorithms and which optimizations they use. Each row stands for a binary parameter.

TPS: Uses the heuristic for interesting pairs.

NRC: States if NRC points are used to skip SMT calls.

RED.: Whether to use the simple redundancy test during the computation of the expansion to convex regions.

NEWNRC: The algorithm will add a NRC point for each SMT solution.

LP SOLVER: States if the algorithm uses the rational LP solver only when needed.

GREEDY: Uses the greedy approach for unsolvable LPS.

We state two different tables, again Table 5 is computed over the first test set, while Table 6 is computed over all instances. In Table 5 we evaluate 674 instances, where the average size of the interpolant is around 17, compared to Table 6 where we evaluate 6998 instances, and an average size of the interpolant of 8. Both tables state the average numbers of the following attributes: the runtime, the total number of timeouts (# to), the absolute number of linear constraints used in the interpolant (# lcs), the relative number of linear constraints compared to the constraint minimization (rel. # lcs), the number of SMT problems we do not need to solve (# skipped SMTs), the number of variables we did not need to introduce to the LP (# skipped LP vars), the total number of instances where we improved the size of the interpolant (# improved), the total number of instances where the method computed the relative minimum compared to all other methods (# rel. minimal), and the number of successful greedy reductions (# suc. greedy reductions).

We will now go through all of the binary parameters and discuss their impact on the performance of the algorithm.

Table 4: Parameters Used in the Different Algorithms

	TPS	NRC	RED.	NEW NRC	LP SOLVER	GREEDY
A_1	0	1	1	1	0	0
A_2	1	1	1	1	0	0
A_3	0	1	1	1	1	0
A_4	1	1	1	1	1	0
A_5	1	1	0	1	0	0
A_6	1	0	1	1	0	0
A_7	1	1	1	0	0	0
A_8	0	1	1	1	1	1
A_9	1	1	1	1	1	1
A_{10}	0	1	0	1	1	0
A_{11}	1	1	0	1	1	0
A_{12}	0	0	1	1	1	0
A_{13}	1	0	1	1	1	0
A_{14}	0	1	1	0	1	0
A_{15}	1	1	1	0	1	0
A_{16}	0	1	0	0	1	0
A_{17}	1	1	0	0	1	0

Table 5: Comparison of the Runtime and Quality for the First Test Set

	runtime [s]	# to	# lcs	rel. # lcs [%]	# skipped SMTs	# skipped LP vars	# improved	# rel. minimal	# suc. greedy reductions
A_1	252.03	162	14.97	88.72	318.20	40992.46	356	276	0
A_2	71.54	13	15.95	91.84	40.44	5225.92	294	168	0
A_3	150.09	72	14.78	88.65	399.73	53677.91	349	287	0
A_4	42.88	8	15.97	92.09	37.99	4656.27	284	160	0
A_5	92.21	24	15.99	92.05	39.90	0	287	160	0
A_6	122.04	42	15.99	92.03	0	7026.24	292	158	0
A_7	72.80	15	15.98	91.94	37.93	5363.94	290	162	0
A_8	234.11	167	15.09	89.41	n.a.	n.a.	343	254	0.19
A_9	58.52	16	15.96	92.05	n.a.	n.a.	288	157	0.04
A_{10}	179.25	98	14.83	88.92	369.95	0	340	270	0
A_{11}	49.95	7	16.00	92.31	n.a.	0	277	152	0
A_{12}	221.05	160	15.49	90.09	0	29252.87	343	244	0
A_{13}	61.78	9	15.99	92.15	0	n.a.	284	157	0
A_{14}	150.54	92	14.93	88.86	324.09	43661.38	352	270	0
A_{15}	47.17	7	16.00	92.22	36.30	n.a.	282	155	0
A_{16}	170.17	111	14.99	89.29	301.84	0	341	252	0
A_{17}	52.00	11	16.00	92.62	31.82	0	275	150	0

Table 6: Comparison of the Runtime and Quality for all Instances

	runtime [s]	# to	# lcs	rel. # lcs [%]	# skipped SMTs	# skipped LP vars	# improved	# rel. minimal	# suc. greedy reductions
A_1	26.66	162	6.71	77.67	73.83	6258.75	4504	4100	0
A_2	7.98	13	7.04	80.52	20.71	1248.81	3713	2833	0
A_3	15.14	72	6.72	78.24	77.33	7311.47	4346	3867	0
A_4	4.60	8	7.05	80.76	18.59	1129.71	3682	2741	0
A_5	10.22	24	7.04	80.57	20.59	0	3700	2818	0
A_6	12.99	42	7.04	80.49	0	1541.84	3712	2839	0
A_7	8.10	15	7.04	80.53	20.12	1267.06	3709	2828	0
A_8	40.01	354	6.71	77.88	n.a.	n.a.	4429	4143	0.12
A_9	8.43	42	7.01	80.46	n.a.	n.a.	3820	2853	0.05
A_{10}	18.03	98	6.73	78.38	74.47	0	4331	3807	0
A_{11}	5.29	7	7.05	80.84	n.a.	0	3673	2713	0
A_{12}	22.55	161	6.78	78.27	0	5094.94	4338	3874	0
A_{13}	6.52	9	7.04	80.66	0	n.a.	3690	2779	0
A_{14}	15.10	92	6.74	78.35	72.81	6228.52	4342	3819	0
A_{15}	5.02	7	7.05	80.78	18.00	n.a.	3680	2737	0
A_{16}	17.04	111	6.75	78.42	70.70	0	4323	3783	0
A_{17}	5.50	11	7.05	80.78	17.53	0	3672	2711	0

TPS: Since we can construct examples in which the heuristic choice of interesting pairs does not include the one which will lead to a reduction, we cannot expect to gain the same results, but will drop some reduction to gain a better total runtime. Here we compare the results in both tables of several pairs: (A_1, A_2), (A_3, A_4), (A_8, A_9), (A_{10}, A_{11}), (A_{12}, A_{13}),(A_{14}, A_{15}), and (A_{16}, A_{17}). All pairs have in common that the only difference between both algorithms is that the first one tries every pair, and the second one uses the heuristic. The results are similar to what is expected, the quality of the solutions are better for the algorithm that tries to reduce every pair, while the running time is better for the algorithm that uses only a subset of these pairs. As an example if we compare A_1 and A_2, the runtime of A_1 is around 3.3 times higher compared to A_2, and the timeouts of A_1 are significantly higher compared to A_2. On the other hand the relative improvement is better by about 3 %, and the number of improved instances has a difference of roughly 800. Compared to the total amount of instances this means that A_1 can improve the interpolant in 11 % of the cases where A_2 does not improve the interpolant at all. This picture is similar for all of these pairs, where the factor for the average computation time is 3.5, and the relative improvement is about 2.5 %. In overall this optimization is still a matter of choice, since both algorithms have their benefits.

NRC: To evaluate the usage of NRC points to skip SMT-problems we compare the following pairs: (A_2, A_6), (A_3, A_{12}), and (A_4, A_{13}). The expectation is that the result of the algorithm should not differ much, since we will only skip SMT-problems when possible. If a new pair needs to be separated, we have to compute it in both cases by an SMT-problem. Thus the main goal of the optimization was to improve the runtime of the algorithm. The runtime over all examples from the first algorithm to the second algorithm increased by a factor of 1.65. In case of A_3 and A_{12} the amount of timeouts significantly increased from 72 to 160. When comparing the quality of the results this option does not effect them in a significant way. Thus this option increases the runtime without sacrificing the quality.

RED.: Here the main goal was to reduce the number of variables of the LP, and therefore its complexity. Again this should only impact the runtime of the algorithm and not the quality of the solution. For this case we compare the following pairs: (A_2, A_5), (A_3, A_{10}), (A_4, A_{11}), (A_{14}, A_{16}), and (A_{15}, A_{17}). The average increase of runtime for the second algorithm of all pairs is around 1.17. This indicates that this simple redundancy test in fact lowers the overall runtime of the algorithm. The quality of the re-

sults is not effected in a measurable way. Thus we can conclude that for such instances a redundancy test is a viable choice to reduce the computation time. We want to remark here that these are certainly instance-specific optimizations, since linear constraints with the same normal cannot be expected in a random instance.

NEWNRC: Since the option *NRC* was successful, we introduced new NRC points while solving SMT-problems during the optimization. This only makes sense if the flag for using NRC points is used. We compare the following pairs of algorithms: (A_2, A_7), (A_3, A_{14}), (A_4, A_{15}), (A_{10}, A_{16}), and (A_{11}, A_{17}). The first algorithm computes more NRC points, while the second only uses initially computed NRC points. In case of A_{14} and A_{16}, we computed initial NRC points without using them in the heuristic. This option only has a small impact when comparing A_{10} and A_{16}, where the runtime is faster for A_{16} compared to A_{10}. On all other pairs the impact of both the runtime and the quality of the results is barely noticeable.

LP SOLVER: For evaluating this option we compare the following pairs: (A_1, A_3), (A_2, A_4), (A_5, A_{11}), (A_6, A_{13}), and (A_7, A_{15}). The average decrease of runtime for the second algorithm is around 0.55. The impact of the quality of the results is minor. Comparing the absolute difference of the relative improvement of the number of linear constraints we observe a difference of 0.30 %, in relative perspective this is an increase by the factor 1.004. Also when we compare the absolute numbers of improved instances the highest difference can be observed in the first pair, where A_1 improved 158 more instances compared to A_3. Further we can observe that this does not imply that A_1 has a better quality, since A3 computed more relative best interpolants in the first test. We can conclude that the runtime was decreased by a factor of 0.55, while the loss of quality was minimal.

GREEDY: Here we expected an increase in the runtime, since in every verification that the second pair makes the LP unsolvable, we have to compute a third LP in which the second pair switches the sides of the constraint. While this could lead to more reductions, it is most likely that it will increase the runtime by a significant amount. To evaluate the option we compare A_8 to A_3 and A_9 to A_4. The increase of the runtime by using the greedy approach is about 2.2, while the improvement is about 0.30 %. In all instances the greedy option in A_8 improved on average 0.12 pairs that would be otherwise stated as not replaceable. This at least indicates that there are instances where this option can in fact improve the solution, but this is not usually the case for the

test sets we evaluated. Also A_8 and A_9 are the only algorithms that exceeded the time limit on the second test set.

As a remark we want to state that even without a time limit the results of the algorithms can differ due to randomness of the SMT-solvers.
We state that for this dataset, and perhaps for every dataset that is created by models occurring during a model checker run, the two promising candidates A_3 and A_4. Therefore we want to compare those with the related methods. They differ only in the option *Tps*, whether to try all pairs for reduction or only a heuristic subset. As mentioned before this option is a trade-off between quality and runtime.

8.4 EVALUATION COMPARED TO RELATED METHODS

In the previous section we concluded that A_3 and A_4 are the best choices of parameters, where A_3 is faster than A_4, but with the costs of losing quality of the resulting interpolant. Here we want to compare the results of our algorithms to the algorithms stated in the Section 4, namely *Constraint minimization* (CM), *Simple interpolants* (SI), *MathSAT interpolants* (MS), and *Beautiful interpolants* (BI). In our paper [3] we stated that the method of BI is not directly applicable to our problems. In this thesis we adapted the method, such that it can handle Boolean complexity. We implemented the proposed strategies, but the method still exceeded the time limit in almost every case. Since the method constructs a brand-new interpolant we were not able to use the current computation to improve the interpolant at all. Therefore we will leave out BI in the rest of this evaluation.
In Table 7, we state for every algorithm the average number of constraints used in the resulting interpolant (# lcs), the percentage improvement of the results compared to CM (rel. # lcs), the number of improved instances (# improved), the number where the method computed the relative minimal solution (# rel. min.), and finally the total runtime (time) in seconds. In the row of CM "rel. # lcs" is obviously 1, since it is stating the size relative to its own size, "# improved" is by the same reason 0, and "# rel. min." states the number of instances where no method constructed an interpolant better than the one computed by CM. When counting the number of relative minimal interpolants we computed the interpolant by every method that is stated here, including $A_1, \ldots, A_{17}$, and count 1 if the interpolant of the method has the same size as the minimum of all methods. This means that multiple algorithms can count 1 if they computed an interpolant with the same (relative minimal) size. The instances are sorted by the bloating factor $(\infty, 0.05)$, and in which situation they occurred in the model-checking process *(bloated,refinement)*. Since this information is not present for the second test set we do not differentiate in this test set. Finally we state everything for all instances together.

Table 7: Comparison of the Runtime and Quality to Related Methods

	A_3	A_4	SI	MS	CM
bloated, $\varepsilon = \infty$, 63 instances					
# lcs	13.32	18.24	18.35	19.78	20.49
rel. # lcs [%]	63	79	86	95	100
# improved	60	49	40	30	0
# rel. min.	39	12	5	4	2
time [s]	142.36	48.36	230.16	2179.00	8.54
bloated, $\varepsilon = 0.05$, 88 instances					
# lcs	30.82	34.58	34.49	36.58	37.13
rel. # lcs [%]	81	90	93	98	100
# improved	82	65	54	38	0
# rel. min.	53	11	9	6	2
time [s]	444.03	139.87	811.10	2716.67	38.58
refinement, $\varepsilon = \infty$, 384 instances					
# lcs	8.56	8.91	9.54	9.77	9.59
rel. # lcs [%]	91	92	100	102	1
# improved	170	150	81	75	0
# rel. min.	152	114	55	46	191
time [s]	68.77	23.43	65.85	224.23	6.01
refinement, $\varepsilon = 0.05$, 139 instances					
# lcs	22.50	22.66	23.30	23.19	22.86
rel. # lcs [%]	97	98	101	101	100
# improved	37	20	15	6	0
# rel. min.	34	16	9	1	99
time [s]	192.13	32.75	280.72	436.97	14.20
Second Test Set, ε unknown , 6324 instances					
# lcs	5.86	6.10	7.11	7.24	7.36
rel. # lcs [%]	77	80	97	99	100
# improved	3997	3398	1122	624	0
# rel. min.	3576	2578	164	170	2009
time [s]	0.76	0.52	2.86	2.10	0.28
all 6998 instances					
# lcs	6.72	7.05	8.01	8.18	8.29
rel. # lcs [%]	78	81	97	99	100
# improved	4346	3682	1312	773	0
# rel. min.	3854	2731	242	227	2303
time [s]	15.14	4.60	24.05	76.66	1.43

As we observe the bloated instances are most likely not optimal, since out of 151 only 4 instances cannot be improved by any method. On the other hand, in roughly 55 % of the refinement instances CM is still the best interpolant that can be derived by any method, and these numbers are only the instances where the lower bound computation did not already confirm that the interpolant of CM is optimal. In total we observe that about 33 % of all instances could not be improved by any method.
To rate the algorithms we will compare their computational time and the number of linear constraints of their solution. We state that both of our algorithms A_3 and A_4 compute a better interpolant in terms of the number of linear constraints as SI and MS, and this by using less computational time. Both algorithms have a much higher success rate in improving the instance, 55 % and 39 %. In comparison, both SI and MS have a success rate lower than 3.5 %. When comparing the runtime of SI to A_3 and A_4 we observe factors of 1.59 and 5.23. Therefore the best-performing algorithm in terms of quality of the solution is A_3 with an average improvement rate of 78 %, and in terms of computational time it is A_4, which uses 4.60 s as an average computation time. It should be noted that the computational time includes the computation of the initial and the final interpolant.
If we consider instances where any method was able to improve the interpolant that was computed by CM, we will end up with 4695 instances. The improvement rate of A_3, and A_4 then is 92 % and 78 %, while the improvement of SI and MS is still only 28 % and 16 %. In this situation we can also determine, with the use of the computed lower bounds, that in 511 of these instances A_3 computed the global optimal solution, which is about 10.9 % of these cases.

8.5 EVALUATION OF THE DESCRIPTION TEST ALGORITHM

In the experiment we test the algorithm for improving the description size of an interpolant, see Section 7. If we would use this test in a model-checking context we would definitely use a lower threshold to trigger the algorithm, since descriptions with already small representation do not need an optimization. To test the viability of the algorithm instead we have set the threshold for triggering to 8 bits, but we will differentiate further which instances would be triggered when it was set to higher values.
In Table 8 we will therefore differentiate in the columns between the thresholds 8, 16, 32, and 64. The rows are first separated between the test set with all instances used before the lower bound computation, and after the lower bound computation, i.e. the instances we used in the previous sections. In both the row *triggered* states how many times the method is triggered, *success* states in how many of the triggered cases the method succeeded, *runtime* states the runtime

Table 8: Comparison of Different Thresholds for the Description Test Algorithm

threshold	8	16	32	64
all instances				
triggered	8923	5355	3374	2513
success	5178	3438	3299	2468
runtime [s]	6.32	9.37	11.14	10.97
worst_init	45.99	68.83	98.99	115.27
worst_comp	15.38	18.10	18.75	18.00
av_init	21.55	30.78	43.56	48.78
av_comp	9.24	10.47	11.46	11.32
success rate [%]	58.03	64.20	97.78	98.21
CM not optimal				
triggered	6558	3511	3333	2501
success	5117	3402	3267	2464
runtime [s]	6.61	10.75	11.04	10.88
worst_init	56.48	95.55	99.51	115.45
worst_comp	14.90	18.28	18.36	17.72
av_init	26.11	42.09	43.69	48.86
av_comp	9.42	11.23	11.31	11.23
success rate [%]	78.03	96.90	98.02	98.52

used for the algorithm, *worst* states the worst coefficient in any of the linear constraints used in the interpolant, and *average* computes the average over all worst coefficients of the linear constraints used in the interpolant. The last two are both stated for the initial (_init) and the computed (_comp) interpolant. In the last row the *success rate* is stated, so how many of the triggered instances were successfully improved in the description.

When evaluating Table 8, we notice that the success rate improves while increasing the threshold, which was expected because this gives the algorithm more chances to improve the constraint. Also the algorithm needs more time for the computation when increasing the threshold. This is also expected due to the fact that more chances to improve the constraint lead to more computations. The decrease in runtime in the last column can be explained by the fact that the in-

stances will trigger if at least one linear constraint is above the threshold, which means that the instances can differ in the number of constraints they need to improve. Another explanation for this effect can be that the instances with a worst coefficient exceeding 64 bits have a smaller description size than the instances that were triggered by the 32 threshold. This again leads to fewer computations. Overall this method improves the description of almost every interpolant with a worst coefficient exceeding 32 bits, which would be our suggestion for the threshold.
We want to remark that all the instances where we improved the description of the interpolant, when CM was already proven to be optimal, does not contradict the lower bound computation. The lower bound computation will compute the number of fixed constraints and then evaluate if they are enough to separate the two state sets. If this is not the case we can increase the lower bound by one. This one linear constraint is not fixed and therefore the description of this constraint can be improved.
With fixed constraints in mind, we want to add that all instances where the algorithm fails to improve the description for a threshold greater or equal than 16 are instances that arose from the refinement and are therefore most likely fixated. This can also mean that even if there is a free choice in at least one constraint, this constraint does not have to be the one that triggered the algorithm by a coefficient.
At last we just want to give an example of such a reduction in the description. The following linear constraint

$$
\begin{aligned}
&-569791006429867630201352854258908620953672250232117493820895896011145414577082095408 0x_3 + \\
&1589690564747834177656465162907126278243135983970658201765086061435371 92x_4 + \\
&-633101118255396395435197020401579697228283900359630996405159913663398711049171956453 0x_5 + \\
&-195685111906199070521033938566177731846436166370909499629500369808271 0x_8 + \\
&-240578424937049837659076973643778050924192647043127346384634276551451772030381142252259890x_9 + \\
&700473483667392673342190082938621674874204271991121852537092787002364x_{10} \\
&\leqslant -240506015526643673525461062811428824911262218027753902258702016092065278372027896477117225
\end{aligned}
$$

was reduced to

$$-1x_3 + 0x_4 + -1x_5 + 0x_8 + -32767x_9 + 0x_{10} \leqslant -32790\ .$$

9

CONCLUSION

In this part of the thesis we presented one main algorithm with different parameter settings to improve the number of linear constraints used in an interpolant of two state sets, and two application algorithms that use the same algorithmic idea, i.e. one computes lower bounds for the size of such interpolants, and the other improves the rational description of these interpolants instead of the number of constraints. During the evaluation we confirmed that most of the optimization strategies significantly improve the runtime of the algorithm. The option of using only a heuristic subset of reduction pairs gives the user the choice of improve the runtime even more by sacrificing solution quality. We recommended parameter settings for the algorithms A_3 and A_4, using all optimizations introduced in Section 5.2 with the difference that A_3 is not using NRC points to reduce the amount of test pairs for the reduction, while A_4 uses the described heuristic. Through the evaluation with the related algorithm *Simple Interpolants* that was developed exactly for this problem and the general approach of interpolation of *MathSAT*, we confirmed that our two settings A_3 and A_4 outperform both algorithms in terms of solution quality and runtime. The final choice between those two algorithms is a trade-off between faster computation and better solution quality, and therefore depends on the application. Finally we evaluated the algorithm for optimizing the description of an interpolant, i.e. the number of bits that are used by the coefficients of a linear constraint. Although the task is a different optimization problem, the underlying description of the state sets is the same. Therefore we used the same technique as for our main algorithm, i.e. do not try to solve the problem directly for the whole system, but solve it piecewise and build up the part of the problem that is needed for the current task. In the evaluation of our proposed algorithm we differentiated between different thresholds. If the worst coefficient of a linear constraint needs more bits to encode then we triggered our method. When we would choose 16 as a threshold our algorithm was successful in reducing the description in over 97 %. Since this is a separate method it could be performed despite the method which computes the interpolant.
For the main algorithm we want to point out that we evaluated how well the methods perform for the interpolation tasks. As we mentioned the problem is a present subproblem of a model-checking process, where the number of linear constraints is a major factor in computing pre-images. The problem for this subproblem is that the formulated problem is not well designed. In fact if we consider a

method that computes the global optimum for our problem instantly, this could lead to a worse model-checking time in a CEGAR model-checking algorithm. This is due to the fact that these interpolations are abstractions and therefore can lead to refinement steps, which are extremely time consuming. Another worse interpolant with respect to the number of linear constraints would perhaps lead to a model-checking run without any refinement step and therefore would perhaps outperform even the imagined instant interpolation method that computes the global minimum. This does not imply that the quality of the interpolation method is not relevant, in fact Damn et al. [19] showed a significant improvement for the model-checking algorithm when they switched to the constraint minimization. Perhaps we only need to consider more things to interpolate in a better way. If for example we could predict in which regions the state sets would evolve, we could substitute the epsilon bloating to guide the overapproximation. By this way the overapproximation would most likely create fewer spurious counterexamples and therefore trigger fewer refinement steps. Another point we want to make is that as far as we know the refinement does not have to include the last interpolation that was made in that state, this would at least give the interpolation the freedom it needs to achieve better interpolants. If this is a problem we would recommend to simply use the constraint minimization, since redundancy is more efficient and without freedom leads to similar interpolants.

9.1 FUTURE WORK

There are still many things that could be improved or combined for our main algorithm. Still a better heuristic choice of the subset for all pairs might be worthwhile, but this is not a straightforward task. The use of rational LP solvers could still be improved, there are existing techniques that use the optimal basis that was computed by a state-of-the-art LP solver and only compute the last simplex steps in rational arithmetic. Further we could combine both application methods into the main algorithm. In case of the lower bound computation, it might be beneficial to compute not lower bounds but fixated linear constraints in the interpolant, since this would reduce the number of test pairs for the reduction. Also if this is implemented within the model-checking algorithm, this could be only triggered when the interpolation was part of the refinement computation. Otherwise there would be no fixated linear constraints in the interpolant. In case of the description test, this is by now a standalone algorithm that, given an interpolant, tries to improve the description. But when we would combine it with our method the description optimization could be a post-process of every reduction our method performs.
Due to the fact that most algorithms use parallel computation we

want to mention ideas for parallelization. The main loop, where every pair or a subset of these pairs are tested for reduction is not trivially parallelizable, due to the fact that these computations are not independent of one another. On the other hand we mentioned earlier that the main part of these pairs is not even close to be separable at once. In fact most of these pairs are proven to be not separable at once after the second pair of convex regions is added to the problem. This would make room for parallel computation since we could run several of those tests for reduction in parallel, and only need to reevaluate the solution if there were multiple reductions happening at once, and even that can be done by solving a single SMT-problem. If the problem does not yield to unseparated points we could use all reductions, and if it finds unseparated points, we could choose one reduction that is used certainly, and use the found pair of unseparated points to refine the other reductions, where also the interpolant constraints needed to be updated. Another point for parallelization would be the greedy approach, where both possible choices could be computed in parallel. The binary blow-up of instances should not happen, due to the fact that in later iterations one of the cases is most likely infeasible.

Part II

PROBABLY APPROXIMATELY CORRECT LEARNING FOR BOUNDED LINEAR ARRANGEMENTS

10 INTRODUCTION

10.1 MOTIVATION

In machine learning a classifier is typically measured by its observed accuracy. While there are different approaches to take several things of the dataset into account, like the distribution of the samples over the classes, the typical proceeding is to split the sample points into a training and a test set. Thus the error between the classifier and the real function is usually the estimation based on the test set. The definition of PAC learning, introduced by Valiant in [54] tried to change this approach. He defined a notion such that a concept is PAC learnable, if there exists an algorithm such that for every given probabilities δ and ε, with probability at least $1 - \delta$ the algorithm outputs a hypothesis concept, satisfying that the true error of this hypothesis is lesser than ε.
In our case we were driven by the first part of the thesis into concepts that are describable by linear constraints, i.e. linear arrangements.

10.2 CONTRIBUTIONS AND OBJECTIVES

Our main objective was to develop an algorithm that is a PAC learning algorithm for a subset of concepts of linear arrangements. While this is impossible for the concept of all linear arrangements, due to the unbounded VC-dimension, we achieved to propose an algorithm that is a PAC-learner for the concept class of bounded linear arrangements in fixed dimension. After proving that our proposed algorithm is indeed a PAC-learner, we found a similar statement that was published in [12]. Since our results differ in certain aspects, and our proof techniques differ a lot, we will do our best to show the different proofs and work out the differences.
Another objective was to show the applicability of our approach. Since we found no indication that the approach from [12] was implemented or would be applicable to normal datasets, this is the second objective of this part of the thesis.
For the applicability of our approach we modeled the basic approach as an ILP. Since the model uses a huge amount of variables the standard approach of using a Branch-and-Bound algorithm would lead to unacceptable runtimes. Therefore we used a CG approach to solve this ILP, which basically creates variables that improve the solution during the runtime and thus can handle the huge amount of variables.

Finally we will introduce a randomized technique to find a meaningful representation of the dataset with a small amount of chosen samples.

10.3 OUTLINE

The remainder of this part of the thesis is structured as follows.
In Chapter 11 we give a very brief introduction to the field of machine learning, while focusing on (agnostic) PAC learning. Furthermore we will introduce the concept of ILP and commonly used techniques to solve them. Here we focus on the concept of column generation.
In Chapter 12 we formally introduce the problem of this part, i.e. the PAC learning problem or more explicit the concept class of bounded linear arrangements in fixed dimension.
Then we will introduce the related work in Chapter 13, where we focus on the proof given by [12], who achieved a similar theoretical result. We will not only sketch the proof but will point out the differences between our problems and our proof-techniques.
Finally we will introduce our algorithms in Chapter 14. First we discuss several adaptions that would be necessary to apply the algorithm of the first part of the thesis to the classification context. This lead to a heuristic approach and therefore was not used afterwards, but is stated for completeness. Then we introduce a basic algorithm, for which we formally prove that it is indeed a PAC-learner in Section 14.3. Afterwards we focus on the applicability and propose the ILP and discuss certain aspects of the CG approach in Section 14.4.
Chapter 15 presents the experiments we made to evaluate our different strategies, and finally compare our algorithm with algorithms typically used in machine learning.
Chapter 16 presents our conclusion of this part of the thesis and will present some ideas for future work.

11

PRELIMINARIES

This chapter is an extension of Chapter 2. We will give a short introduction to machine learning and go into more details about PAC learning. After this we will briefly introduce ILP, and a concept how to solve ILPs if they have a large number of variables called column generation. The content of the chapter is not original and is mentioned for completeness.

11.1 MACHINE LEARNING

A typical supervised classification task is given by a set of samples in X and their class. In prototypes of real-world classification tasks, X is a combination of continuous and numerical features. Throughout this work we will consider that $X = \mathbb{R}^d$, where d is an arbitrary but fixed dimension. The number of possible classes is numerical, i.e. $\{1, \dots, f\}$. A learning algorithm A computes for a set of samples $S = \bigcup_i S_i$, where each $S_i \subset X$ consists of samples of the class i, a *hypothesis* $A(S)$, which is a function from $\mathbb{R}^d$ into $\{1, \dots, f\}$. The task is to correctly classify new samples, drawn by the same distribution $\mathcal{D}$ as the set S, correct. The *true error* of a hypothesis $H : \mathbb{R}^d \to \{1, \dots, f\}$, is therefore

$$\mathrm{Err}_{\mathcal{D}}(H) := \mathrm{Pr}_{(x,x_f)\in\mathcal{D}\mathbb{R}^d\times\{1,\dots,f\}}[H(x) \neq x_f] \,, \tag{47}$$

which is all with respect to the used distribution $\mathcal{D}$. In practice $\mathrm{Err}_{\mathcal{D}}$ is not known, and typically the error rate due to a test set is used as an estimation. The true error therefore depends not only on the algorithm A itself, but on the size of S and on its underlying distribution $\mathcal{D}$.

When evaluating machine learning algorithms on real-world tasks different statistical indicators are used. Typically a cross-validation, or a k-fold, is computed for a dataset. This is a technique where we divide the whole sample set into k disjoint subsets with almost the same size. These sets have to be created randomly, so without manually creating simple subsets. For example not only the size of the subsets should be almost equal, but also the distribution of the classes should be almost the same compared to the whole sample set. Then every algorithm we want to compare is trained k times in total. Every time only $k-1$ of these sets are used to train the machine learning algorithm, while the last set is always used to test the then resulting classifier. Therefore every point in the sample was present as a new point in one test, and was used $k-1$ times to train the classifier. The results of the algorithm then can be evaluated by comparing, for example the *classification accuracy*. This measures the relative factor of how many points of the

sample were correctly classified. Other factors are for example *false negatives* and *false positives rates*, which give a more detailed view on the classification accuracy. False negatives are the errors, where the classifier stated that a condition does not hold, while in fact it does. False positive are in contrast where the classifier indicates that it is in a specific class, while the sample is not part of that class. In certain classifier problems it is crucial that one of these errors is small, and one might prefer a classifier which has an overall worse error rate, when for example the false positive rate is small.
In the next sub-sections we will describe the ideas behind PAC learning and agnostic PAC learning and how they differ from each other.

11.1.1 *Probably Approximately Correct Learning*

The content of this section is based on [35]. In PAC learning theory *a priori* knowledge of the problem is used, in fact the target function which determines the class for each sample is limited to a set of concepts, called *concept class*. Additionally the algorithm itself can decide how many samples it will need. Both of these conditions are the main points, why it is doubtful that the theory is relevant for real-world tasks.

Definition 27 (PAC learnable)**.**
Let C *be a concept class over the feature space* $\mathbb{R}^d$*. We say that* C *is* PAC learnable *if there exists an algorithm* A *with the following property: For every concept* $c \in C$*, for every distribution* $\mathcal{D}$ *on* $\mathbb{R}^d$*, and for all* $0 < \varepsilon < 1/2$ *and* $0 < \delta < 1/2$*, if* A *is given the inputs* ε *and* δ *it chooses the size of* S*, then with a probability at least* $1 - \delta$*,* A *outputs a hypothesis concept* $H \in C$ *satisfying* $\mathrm{Err}(H) \leqslant \varepsilon$*. This probability is taken over* $\mathcal{D}$ *and any internal randomization of* A*.*

If the runtime of L is polynomial in $1/\varepsilon$ and $1/\delta$, we say that $\mathcal{C}$ is *efficiently* PAC learnable. Therefore a PAC learning algorithm has additional inputs to the samples, namely the two probabilities ε and δ. Although the first probability (δ) was present all throughout the history of machine learning, which is the error of the predicted classifier due to future sample points, the second probability (ε) was first introduced to machine learning by the concept of PAC learning by Valiant in [54]. This probability is the only way to prove that a given machine learning algorithm fulfills a certain given threshold for the first probability on a given problem.
To get an idea of what a PAC learning algorithm looks like, and since we will use a similar technique to prove that our algorithm is a PAC-learner, we give a simple example of the rectangle learning game.

11.1.1.1 *Rectangle Learning Game*

The goal of this one-person game is to learn a rectangle by drawing samples. The target rectangle R is axis-aligned and only two-dimensional. A positive sample indicates that the point is in the rectangle, a negative sample indicates that the point is outside of the rectangle. The player has to decide how many samples he needs to predict a hypothesis rectangle R', which should be a close approximation of R. To check how good the approximation is we compute the

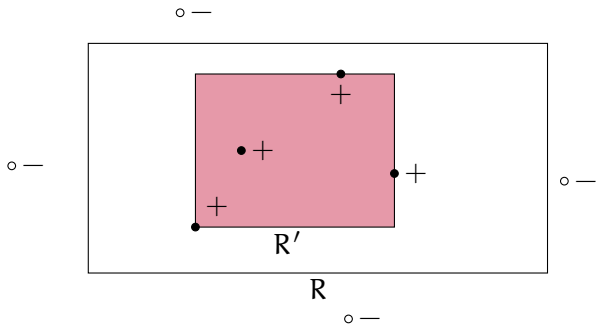


Figure 11: Running example for Rectangle Learning Game

probability that a point is drawn out of the region $(R \setminus R') \cup (R' \setminus R)$, which is basically the sum over false negatives and false positives. One central point is that we do not know what distribution we take over the points in $\mathbb{R}^2$, therefore we assume that the distribution is arbitrary but fixed over the whole process. So every sample that is drawn by the player, and the error of the hypothesis rectangle R' are all computed over the same distribution $\mathcal{D}$.
The player decides to choose the strategy to compute always the tightest axis-aligned rectangle around the samples, if no positive point is in the sample he will return $R' = \emptyset$.
Now we will show that for any input $(R', \varepsilon, \delta, \mathcal{D})$, we can predict a sample size n such that with a probability at least $(1 - \delta)$ the tightest axis-aligned rectangle R' has at most an error of ε, where all samples and the error are computed over $\mathcal{D}$.
First we can observe that $R' \subset R$, since we choose the tightest axis-aligned rectangle around the positive samples. This means the error of R' is only the sum of false negatives, formally

$$error_{\mathcal{D},R}(R') = Pr_{x \in \mathcal{D}}[x \in R \setminus R'] \tag{48}$$

Further we describe $R \setminus R'$ by a union of four stripes, one on each side of the rectangle. When we are able to state that each of those stripes has an error smaller than $\varepsilon/4$ we can conclude that the complete error is smaller than ε.
Therefore we take a look at the upper strip T'. Additionally we define the rectangular strip T at the top of R, such that T has exactly the weight $\varepsilon/4$ under the distribution $\mathcal{D}$. This is possible due to the fact

that $\mathcal{D}$ is a continuous distribution. It is clear that T' has a weight greater than $\varepsilon/4$ if and only if T' includes T, since by definition they have a subset-superset relation. Furthermore, T' includes T if and only if no point in T appears in the sample S. This statement holds, since if there is a point $p \in T$ in the sample, then this must be a positive sample. Therefore the rectangle R' must include this point, and the strip T' must be smaller than T. We now want to compute

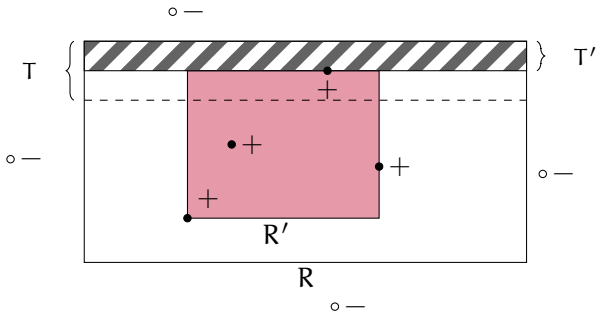


Figure 12: Error of one stripe T' in the Rectangle Game

how likely it is that every drawn sample misses the region T. This is by definition of T exactly $(1-\varepsilon/4)^n$, when n is the size of samples. Which concludes that T' has a probability greater than $\varepsilon/4$ with a probability $(1-\varepsilon/4)^n$. The same analysis is true for the other three stripes, which leads to the following. The error of R' is greater than ε with an estimated probability of $4\,(1-\varepsilon/4)^n$. The estimation is due to the fact that we counted every corner two times. With this estimation we can compute the sample size based on ε and δ, i.e.

$$m \geqslant (4/\varepsilon)\ln(4/\delta)\ . \tag{49}$$

11.1.2 *Polynomial Learnability with Respect to Concept Complexity*

In this section we will introduce the theoretical concept of polynomial learnability (poly-learnable), developed by Blumer et al. [10]. This concept is used in the related paper [12] and is the main difference of their statement and the statement we prove in this thesis. They first defined a triple for a concept class C. They fix the learning domain such that X is in finite dimension d, in our case $\mathbb{R}^d$, C be the concept class on X and *size* be a function from C to $\mathbb{Z}^+$ called the *concept complexity measure*.

Definition 28.
Let (C, X, size) *be as defined above. It is called* (properly) polynomial learnable (poly-learnable) *if there exists a polynomial-time algorithm* A *that takes as input a sample of a concept in* C*, outputs a hypothesis in* C*, and has the property that for all* $0 < \varepsilon, \delta < 1$ *and* $s \geqslant 1$ *there exists a*

sample size $m(\varepsilon, \delta, s)$*, polynomial in* $1/\varepsilon$, $1/\delta$*, and* s*, such that for all target concepts* $c \in C$ *with* $\mathrm{size}(c) \leqslant s$*, and all probability distributions* P *on* X*, given a random sample of* c *of size* $m(\varepsilon, \delta, s)$ *drawn independently according to* P*,* A *produces, with a probability at least* $1 - \delta$*, a hypothesis* $h \in C$ *that has an error of at most* ε*. The smallest such* $m(\varepsilon, \delta, s)$ *is called the sample complexity of* A*.*

Note that the concept class C itself can have an infinite VC-dimension. Further they use the notion of the *effective hypothesis space of* A *for target complexity* s *and sample size* m $C^A_{s,m}$ that is basically the A–image, i.e. all hypotheses produced by A, of all possible m–samples of concepts $c \in C$ such that $\mathrm{size}(c) \leqslant s$.
Additionally they use the notion of an *Occam algorithm* for C.

Definition 29.
Given the triple (C, X, size)*, an algorithm* A *is an* Occam algorithm *for* C *if there exists a polynomial* $p(s)$ *and a constant* α*,* $0 \leqslant \alpha < 1$*, such that for all* $s, m \geqslant 1$*, the* VC*-dimension of* $C^A_{s,m}$ *is at most* $p(s)m^\alpha$*.*

Finally they state a theorem that combines poly-learnable with Occam-algorithms.

Theorem 30.
Let A *be an Occam algorithm for* C *with effective hypothesis space* $C^A_{s,m}$ *for target complexity* s *and sample size* m*. Then if the* VC*-dimension of* $C^A_{s,m}$ *is at most* $p(s)(\log m)^l$ *for some polynomial* $p(s) \geqslant 2$ *and* $l \geqslant 1$*, then* A *is a polynomial-time learning algorithm for* C *using the sample size*

$$m = \max\left(\frac{4}{\varepsilon}\log\frac{2}{\delta}, \frac{2^{l+4}p(s)}{\varepsilon}\left(\log\frac{8(2l+2)^{l+1}p(s)}{\varepsilon}\right)^{l+1}\right).$$

11.1.3 *Agnostic Probably Approximately Correct Learning*

Agnostic PAC learning is a variant of PAC learning. Due to [31] and [34], this is a more relevant concept to practical machine learning, since no a priori knowledge is used. The samples are here drawn by a distribution over the product of $\mathbb{R}^d \times \{1, \ldots, f\}$, and thus there exists no target concept c nor a concept class C. Another difference to PAC learning is the goal of the algorithm A. In PAC learning we have to find a hypothesis with an arbitrarily small true error. In agnostic PAC learning theory we only need to find an arbitrarily small true error in the limit of our hypothesis class $\mathcal{H}_d$.

Definition 31 (Agnostic PAC learning)**.**
An algorithm A *is an* agnostic PAC learning algorithm*, if there exists a function* $n : \mathbb{R}^2 \times \mathbb{N} \to \mathbb{N}$ *that is bounded by a polynomial, such that for any given* $\varepsilon, \delta > 0$*, any* $d \in \mathbb{N}$*, for any distribution* $\mathcal{D}$ *over* $\mathbb{R}^d \times \{1, \ldots, f\}$*, and any set* S *of size* $n(1/\varepsilon, 1/\delta)$ *drawn by* $\mathcal{D}$*, the error*

$|\mathrm{Err}_{\mathcal{D}}(A(S)) - \inf_{H \in \mathcal{H}_d} \mathrm{Err}_{\mathcal{D}}(H)| \leqslant \varepsilon$ *with probability greater-or-equal* $1 - \delta$. *The probability is taken over* $\mathcal{D}$ *and any internal randomization of* A.

Also in this theory A is an *efficient* agnostic PAC learning algorithm if the running time of A is polynomial in $1/\varepsilon$, $1/\delta$ and in the (bit-length) size of the representation of S.

11.1.4 *Main Results for (Agnostic)* PAC*-Learning*

The main results in the field of PAC learning were achieved by combining the theory with the notion of the VC-dimension, introduced by [55].

Definition 32 (VC-dimension)**.**
Given a nonempty concept class $C \subseteq 2^X$ *and a set of points* $S \subseteq X$, $\Pi_C(S)$ *denotes the set of all subsets of* S *that can be obtained by intersecting* S *with a concept in* C *that is,* $\Pi_C(S) = \{S \cap c : c \in C\}$. *If* $\Pi_C(S) = 2^S$, *then we say that* S *is* shattered *by* C. *The* Vapnik–Chervonenkis (VC) dimension *of* C *is the cardinality of the largest finite set of points* $S \subseteq X$ *that is shattered by* C. *If arbitrary large sets are shattered, the* VC*-dimension of* C *is infinite.*

A combination by [10] led to the theorem that every concept class C is PAC learnable if and only if the VC-dimension of the C is finite.
For agnostic PAC learning theory [31] proved that if an algorithm misclassifies a minimal (with respect to its hypothesis class) number of samples efficiently, and the VC-dimension is finite, then the algorithm is an efficient agnostic PAC learning algorithm.
There are also some negative results for (agnostic) PAC learning. It was shown that for arbitrary dimensions the concept of intersection of hyperplanes is not efficiently PAC learnable, unless $P = NP$, even if the number of hyperplanes is restricted to two [9].

11.2 INTEGER LINEAR PROGRAMMING

This section is a continuation of Section 2.2, where we introduced linear programming. In integer linear programming we still consider that a linear function is optimized over a set of variables that is described by linear constraints. The difference is that specific variables are here forced to be integer instead of real valued in case of linear programming.
To obtain an ILP in standard form, we consider $A \in \mathbb{Q}^{m \times n}$, $B \in \mathbb{Q}^{m \times k}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, $d \in \mathbb{Q}^k$, $x \in \mathbb{Q}^n$, $y \in \mathbb{N}^k$ and

$$\begin{aligned} z_{ILP} = \min \quad & c^T x + d^T y \\ \text{s.t.} \quad & Ax + By = b \\ & x, y \geqslant 0 \\ & y \text{ integer.} \end{aligned} \tag{50}$$

Notice that similarly to linear programming problems we can model inequality constraints by adding slack or surplus variables. To highlight the appearance of continuous and discrete variables, 50 is also known as a mixed integer linear program. In the case where y only takes binary values the problem is called Binary Integer Linear Program (BILP).
In contrast to linear programming problems, integer linear programmings problems are more difficult to solve. In fact already a BILP is NP-hard, proofs are given by Schrijver [50] and are based on Cook [14] and Karp [33].
Due to this fact there are typically three kinds of algorithms to solve ILPS. Exact algorithms that terminate with the global optimal solution but may take an exponential number of iterations. Approximate algorithms that compute a suboptimal solution in polynomial time together with a bound on the degree of sub-optimality. Heuristic algorithms that provide suboptimal solutions in polynomial time, but without any guarantee on their quality.
We will focus on a specific exact algorithmic idea, which is based on the most common one to solve ILPS the branch and bound framework.

11.2.1 *Branch and Bound*

Branch and bound is a "divide and conquer" approach to explore the set of feasible integer solutions, it was first proposed by Land and Dog [37] in 1960 and later extended, implemented and tested by Dakin [17] in 1965. It is based on the idea to use bounds on the optimal objective value to skip divided subspaces that cannot improve the current best feasible value. Let F be the set of all feasible solutions of an ILP of the form

$$\begin{aligned} z_{OPT} = & \min c^T x \\ & \text{s.t.} x \in F \, . \end{aligned}$$

Now consider a partition of F into a finite collection, such that $F = \bigcup_{i \in I} F_i$, which then are solved separately.

$$\begin{aligned} z_i = & \min c^T x \\ & \text{s.t.} x \in F_i \end{aligned}$$

Then it would hold that $z_{OPT} = \min\{z_i | i \in I\}$. Since these problems may be almost as difficult as the original problem we can consequently partition the subproblems themselves, leading to a tree of subproblems. This is the branching part of the method. We additionally assume that there exists an algorithm to compute a lower bound $lb(i) \leqslant z_i$ in a tolerable time for each of those subproblems. A popular method to compute lower bounds is to solve the LP relaxation,

i.e. ignore the integrality constraints of all variables. If a subproblem is solved to optimality we get an upper bound $ub \geqslant z_{OPT}$ of the initial objective value.
The essence of this procedure, or the bounding part, is that if we encounter a lower bound $lb(i)$ for a subproblem that is greater than the best ub we found so far, then we do not need to consider this subproblem anymore since the optimal solution of this subproblem is worse than the one already encountered in the process.
There are several strategies how to proceed in a branch and bound algorithm, i.e. which subproblems should be computed next and how subproblems are created. Two extreme ways of choosing the next subproblem are "breadth-first search" and "depth-first search" that prefer subproblems of the depth, and respectively subproblems along each branch.
A common way to partition a problem is to use the optimal fractional solution x^* of the LP relaxation, and adding for a fractional integer variable x_i^* either $x_i \leqslant \lfloor x_i^* \rfloor$ or $\lceil x_i^* \rceil$ to the current constraints.

11.2.2 *Column Generation Approach*

Column generation is used when solving LPs or ILPs with a large number of variables. We will first introduce the concept of a column generation approach for LPs and will afterwards introduce the Branch and Price Framework which combines column generation with a branch and bound framework to solve ILPs.
The idea behind column generation is to solve a LP by iteratively adding the variables of the model, since only a tiny fraction of the variables is needed to prove optimality. So consider the LP we want to solve, this is called the Master Problem (MP). In the beginning the problem only consists of a restricted number of variables, therefore the problem is called Restricted Master Problem (RMP). After solving the RMP to optimality the values of the dual variables are known, due to the fact of strong duality. Similarly to the procedure in the simplex method from Section 2.2.2, we now search for a variable that is not currently in the basis of the LP, such that its reduced cost in the current basic solution is negative. This subproblem which searches for new variables is called *pricing problem*.
If the pricing problem found a variable with reduced cost, the RMP is then updated with a new column and solved again. When the pricing can prove that no variable with reduced cost exists, it guarantees that the current solution of the RMP is the optimal solution for the MP.

11.2.2.1 *Algebraic Derivation*

Consider the linear program,

$$\begin{aligned} \min \quad & \sum_{j \in J} c_j \lambda_j \\ \text{s.t.} \quad & \sum_{j \in J} a_j \lambda_j \geqslant b \\ & \lambda_j \geqslant 0, j \in J \end{aligned} \tag{51}$$

with $|J| = n$ variables and m constraints. We assume that working with (51) is not an option because n is large. Instead we consider the RMP which contains only a subset $J' \subset J$ of variables. An optimal solution λ^* to the RMP does not need to be optimal for the MP. Denote by ϕ^* an optimal dual solution to the RMP, similarly to the pricing step in the simplex method, we search now for a non-basic variable of negative reduced cost to enter the basis. In our case we therefore have to solve the Pricing Problem (PP)

$$v(PP) := \min\{c_j - \phi^* a_j \mid j \in J\}. \tag{52}$$

If $v(PP) < 0$, we will add the variable λ_j corresponding to the optimal value with its column of coefficients (c_j, a_j) to the RMP. Now we iterate the process, by solving the RMP again to optimality and obtaining the optimal dual variables, until we obtain $v(PP) \geqslant 0$ in one iteration. In this case λ^* is also the optimal solution for (51). The correctness and finiteness of the method is inherited by the simplex method, see Section 2.2.2 for details.
Typically the problem (52) is an optimization problem itself that can be solved easily.

11.2.2.2 *Branch and Price*

When solving an ILP by branch and bound, the linear relaxation is typically used to compute lower bounds for the subproblem. If we solve the linear relaxation by column generation in each node one speaks of a branch and price framework.
There are many aspects that will change the performance of a branch and price framework. The column generation suffers from the *tailing-off effect* [28], which describes the slow convergence to the optimum. A main reason for this effect lies in the unstable behavior of the dual variables. The dual solutions of consecutive steps may be far apart from each other. While there are some techniques to reduce this effect, it is also an option to branch earlier. This *early branching* will leave a node without computing the optimal solution. Therefore we have to compute the lower bound in a different way. Assume there is an upper bound $\kappa \geqslant \sum_{j \in J} \lambda_j$ that holds for an optimal solution of the MP, then in each iteration the optimal function value $\bar{z}$ of the RMP

will lead not only to an upper bound of z^*, but also to a lower bound. Since we cannot reduce $\bar{z}$ by more than κ times, the smallest reduced cost $\nu(PP)$ is, hence

$$\bar{z} + \nu(PP)\kappa \leqslant z^* \leqslant \bar{z} \,. \tag{53}$$

So to compute a lower bound without solving the MP to optimality for the branch-and-price framework, we can use the highest lower bound computed by (53) over all iterations where the PP was solved optimally.

11.3 LINEAR ARRANGEMENTS

We want to introduce some basic theorems known by the arrangements of hyperplanes, also known as *linear arrangements*. A finite hyperplane arrangement $\mathcal{A}$ is a finite set of hyperplanes in some vector space $V \cong K^n$. We are only interested in the case where $K = \mathbb{R}$. In this case a hyperplane is a linear constraint with equality used as a sign, as we defined it in Section 2. An arrangement $\mathcal{A}$ is in *general position*, if

$$\{H_1, \ldots, H_p\} \subseteq \mathcal{A},\ p \leqslant n \Rightarrow \dim(H_1 \cap \cdots \cap H_p) = n - p$$
$$\{H_1, \ldots, H_p\} \subseteq \mathcal{A},\ p > n \Rightarrow H_1 \cap \cdots \cap H_p) = \emptyset.$$

A region of an arrangement $\mathcal{A}$ is a connected component of the complement X of the hyperplanes:

$$X = \mathbb{R}^n - \bigcup_{H \in \mathcal{A}} H \,.$$

Usually the set of regions is denoted by $\mathcal{R}(\mathcal{A})$, and

$$r(\mathcal{A}) = \#\mathcal{R}(\mathcal{A}) \,,$$

is the number of regions of $\mathcal{A}$.
In our case the algorithms calculate a set of linear constraints L, where the inequality is not allowed to hold with equality. This set can be interpreted as a linear arrangement, but we will define a region similar to the enlargement of points to convex regions in Section 5.1.2. A region r of a set of linear constraints L is defined as a convex region that is not distinguishable by any linear constraint in L. Similar the set of all regions is denoted by $\mathcal{R}(L)$.
Our classification will be a function over the regions defined over the set of linear constraints L, $c : \mathcal{R}(\mathcal{L}) \to \{1, \ldots, f\}$, based on a previously computed set of linear constraints L. This can of course be interpreted as a function with the domain $\mathbb{R}^n$. Note that the function over the regions defined by the linear arrangement would be over $X \subset \mathbb{R}^n$, which needed to be enlarged to $\mathbb{R}^n$.

To calculate the actual number of regions of an arrangement is a non-trivial task, but we will later only need an upper bound to estimate the probabilities in our proof. Therefore we can use a part of a proposition about arrangements in general position, see Proposition 2.4 in [52].

Proposition 33 (general position)**.**
Let $\mathcal{A}$ be an arrangement over $\mathbb{R}^n$ with m hyperplanes in general position. Then the number of regions are

$$r(\mathcal{A}) = 1 + m + \binom{m}{2} + \cdots + \binom{m}{n} .$$

Since there exists no arrangement of m hyperplanes with more regions, this is an upper bound for all arrangements over $\mathbb{R}^n$ with m hyperplanes. Because our actual classifier is based on another definition of regions, we show that in the general case the two sets are of the same size.

Lemma 34.
Let L be a set of linear constraints where every constraint in L is not an equality constraint, and $\mathcal{A}$ be the corresponding linear arrangement. If $\mathcal{A}$ is in general position, then it holds that

$$\#R(L) = \#R(\mathcal{A}) .$$

Proof. It is trivial that every region in $R(\mathcal{A})$ has a corresponding region in $R(L)$. In case that $\mathcal{A}$ is in general position though it is also true that there is no other region $R(L)$. So consider that there is a region that is in $R(L)$, but not in $R(\mathcal{A})$. This region has a lower dimension, since it must be a convex subset of the complement of X, which can be at most $n-1$. To achieve that the dimension of a region in $R(L)$ is at most $n-1$, at least two linear constraints must face each other to form an equality. This contradicts the condition that $\mathcal{A}$ is in general position, since then those two corresponding hyperplanes would lead to

$$\dim(H_1, H_2) = n - 1 \neq n - 2 .$$

□

This leads to the fact that the upper bound given by Proposition 33 holds also for $R(L)$.

12 PROBLEM DEFINITION

We want to formulate the problem that is solved by the exact algorithm without using the concept of column generation.
The problem is a supervised classifier problem, whose general form is described in Section 11.1. We will focus on problems where the features are purely real, i.e. $X = \mathbb{R}^d$, and all samples are labeled. Also the number of classes is not bounded to be binary. As in general the task of the algorithm A is to compute a hypothesis or a classifier $A(S)$ based on the samples S.
The hypothesis/classifier in all of our algorithms is defined by a function, whose domain are the regions of an arrangement. So basically the algorithms compute a set of linear constraints, which can be interpreted as a linear arrangement, and the regions of this arrangement are then used to define the classifier. In the learning phase where no learning-error occurred, which is the usual case for our exact algorithm, the classifier will map the regions accordingly to the classes of the learning points.
Additionally we want to state that our exact algorithm is a PAC-learner, therefore we have to specify for which concept class this holds. Accordingly we define $\mathcal{C}_k^d$, which consists of concepts in $\mathbb{R}^d$ defined by k hyperplanes that induce an equivalence relation, i.e. two points in X are equivalent if and only if none of the k hyperplanes can separate them. Note that d and k are bounded for all concepts in $\mathcal{C}_k^d$.

13

RELATED WORK

In this chapter we want to focus on presenting the algorithm introduced by [12]. As we already mentioned, the theoretical result we will give in Section 14.3, is similar to the one stated in this paper. Their algorithm and therefore the proof differs from ours and the paper was not known to us when we made the proof ourself.
They introduce the exact same concept class $\mathcal{C}_k^d$, but since they will use the concept of *poly-learnable* (11.1.2) they will only fix the dimension d and for the number of hyperplanes they will consider it to be bounded for the target concept. Their own hypothesis will in fact use $k \ln m$ number of hyperplanes.
The proposed algorithm will do the following. First they draw the samples. Then they compute a subset Q of all possible hyperplanes in S that includes a subset of s hyperplanes that separates the samples of different classes. After that they introduce a graph, where each vertex is a sample point and for every sample point of the other class an edge is introduced between the vertices. A hyperplane will cover an edge, when the hyperplane separates the corresponding sample points. When $\mathcal{F}$ is a set of hyperplanes, then it is easy to see that they have to solve a set-covering problem on this graph.
They will not solve the set-covering problem optimally, but use an approximation algorithm with ratio $O(\lg m)$, and thus the hypothesis of the algorithm uses at most $O(s \lg m)$ hyperplanes.
To reduce the number of all hyperplanes to a finite subset they use an argument of the geometrical dual space. First they introduce equivalence classes of hyperplanes such that two hyperplanes are equivalent if all samples are identical under both hyperplanes. Then they use the fact that every point in the primal space is a hyperplane in the dual space and vice versa. Thus every point of the sample p is mapped to a hyperplane $g(p)$. Therefore each equivalent class is represented by the interior of a $(d-k)$-dimensional face of the arrangement consisting of all hyperplanes $g(p), p \in S$, where k is the number of points of S on a hyperplane in the equivalence class. Accordingly the number of hyperplanes is at most $O(m^d)$. By adding all hyperplanes to $\mathcal{F}$ it leads to the fact that the optimal solution is s, and therefore the approximate solution by the greedy algorithm will be $O(s \ln m)$.
Their time complexity is dominated by the greedy algorithm, which takes $O(\sum_{p \in \mathcal{F}} |p|)$, where $|p|$ is the size of the set p. Since the set represents the edges its covers, the size is bounded by the number of edges in the graph, i.e. $O(m^2)$. Combined with the fact that $\mathcal{F}$ consists of at most $O(m^d)$ sets, this leads to an overall time complexity

of $O(m^{d+2})$.

This, combined with other details, leads to the fact that their algorithm proofs that the concept is *poly-learnable*. The main difference to our algorithmic idea is that their solution is using far more constraints than are needed to classify the data. Their proof is very elegant and based on both the geometrical dual space (see, for example, Edelsbrunner [24]), and the theorems provided by Blumer et al. [10]. Our proof of the fact that the concept class $\mathcal{C}_k^d$ with bounded d and k is PAC learnable is given in Section 14.3.

14 THE ALGORITHMS FOR SUPERVISED LEARNING

In this chapter we will introduce different types of learning algorithms. The first algorithm proposed in Section 14.1 is a theoretical adjustment to our main algorithm of the first part. This adjustment is only theoretical, since the adjustments where not able to turn this approach into a PAC learning algorithm. Further we will introduce the basic algorithm in Section 14.2. In the following Section 14.3 we prove that the basic algorithm is a PAC learning algorithm for the concept class $\mathcal{C}_k^d$ with bounded d and k. In Section 14.4 we propose the ILP that models the approach of the basic algorithm, and introduce all problems for the CG-approach, i.e. the master problem, and the pricing problem. Finally in Section 14.5 we will introduce a randomized approach handling the sample size. Instead of taking all samples into account, this approach will solve several smaller samples and increase the amount of samples each time.
As noted before, despite the algorithmic approach our output classification is always based on a set of linear constraints L computed by the algorithm. The actual classifier is then computed by mapping the regions $R(L)$ to the class with the most samples in this region.

14.1 HEURISTIC APPROACH

This algorithm was considered to be a direct application of the algorithm of the first part for the context of machine learning. Therefore we will not describe the whole algorithm again but point out certain aspects that had to be revised to make this work.

14.1.1 *Missing Interpolant*

The algorithm described in Section 5 is based on improving an already existing interpolant, or more formally correct a set of interpolant constraints, therefore an initial interpolant must be computed. In the context of state sets that was always an easy task, since the algorithm could start with either one of the sets of interpolant constraints used in the description of the state sets.
In this case we could use the following approach. Starting with an empty set of constraints, we search for a pair of points that were not separated by any constraint in the set. Then we compute the hyperplane separating this pair and add it to the set. This leads to large sets of interpolant constraints, nevertheless viable interpolant constraints that suffice Proposition 22.

14.1.2 *Samples Instead of State Sets*

In the first part we handled unions of convex regions in an implicit way. Therefore we enlarged points found through SMT-solvers to convex regions to ensure the termination of the algorithm. In this new situation we handle a finite amount of samples, and therefore we do not need to enlarge them to convex regions to ensure termination. This leads to a simpler version of the LP stated in Section 5.1.3.

$$
\begin{aligned}
\max \quad & \varepsilon \\
\text{s.t.} \quad & a^T p^i \leqslant b - \varepsilon \\
& a^T q^i \geqslant b + \varepsilon \\
& a \leqslant 1 \\
& a \geqslant -1 \\
& \varepsilon \geqslant 0
\end{aligned}
$$

Additionally the samples are real-valued and there is no safety criterion present in this problem. So we are able to switch to a state-of-the-art LP solver. This leads to much better performance on this subproblem compared to the algorithm of the previous part. A technical difference is that this LP is always solvable, by setting every variable to zero. Since a touching case as described in Section 5.1.3.1 is not feasible in this context, the circumstance that the objective function is zero indicates that there is no separating linear constraint.

14.1.3 *Allow Misclassification of Points*

From all new aspects that arose in this setting, this is the most complicated one. Basically the first algorithm was developed to separate every single point, since it was created for a safety critical environment. In the CEGAR approach it was essential that we can separate a single counter example that was discovered to make sure that in the next refined circle this is never part of the state set. This stands in absolute contrast to the situation in machine learning, where outliers or simply misclassified samples are well-known problems that are common in supervised learning datasets that are derived by real world applications. When an algorithm ignores this fact it will most likely lead to an over-fitting classifier.
To change the least in the central part of the algorithm, i.e. the LP and the task of finding indistinguishable points, we tried to identify these outliers at best before we start reducing the description of the classifier. Therefore we took the initial interpolant as a reference. An outlier would most likely create a region where it is the only point of its class, therefore we mark these points as possible outliers, and will ignore them in the first iterations of the algorithm.

14.1.4 *Find Indistinguishable Points*

Instead of solving an SMT-problem, like described in Section 5.1.1, we did the following. For every point we compute for every current linear constraint a bit, which is 0 if the point is invalid for the linear constraint and 1 if it is valid. Then we sort all of them by their corresponding bit-vector. Now every point that has the same bit-vector is indistinguishable under these linear constraints, so we just have to check if there are points of different classes with the same bit-vector. If we find such a bit-vector with points of different classes we have to separate them from each other. Note that a bit-vector with more than two classes will always end in a misclassification of one of them, since we only compute one additional linear constraint inside the subroutine described in Section 5.1. Also note that if there are more than two bit-vectors with points from different classes the sides of the linear constraint can differ from the first bit-vector, i.e. a point of class A is decided to be valid under the new linear constraint, then another point of A with another bit-vector must not be forced to be on the same side. This situation is very similar to the one we discussed in Section 5.1.3.2, with the difference that we have this knowledge directly when we compare all bit-vectors. The only idea where we can solve this optimally is where we try every single permutation, which would use a lot computational effort, without gaining the certainty of being global optimal. Therefore we stuck with the same approach as before, where we choose the side for the "pair", in this case all points with the same bit-vector, and then only change the sides of the last "pair" if the LP cannot compute a separating linear constraint.

14.2 EXACT APPROACH

In this approach our goal is not to make a heuristic algorithm but instead to compute a global minimum. Let $k \in \mathbb{N}$ be an input for the algorithm stating the maximal amount of linear constraints used in classifier. The goal of the algorithm is to compute the minimal number of linear constraints that is needed to classify the samples without misclassification.
The main idea of our algorithm is that instead of finding arbitrary linear constraints in $\mathbb{R}^d$, we limit our search to a finite subset of linear constraints based on the samples S. This subset Q should fulfill the following condition.
For a subset $A \subset S$ there exists a hyperplane in Q that separates A from $S \setminus A$ if and only if there exists a hyperplane that separates A from $S \setminus A$.
For simplicity we begin to consider that S is in general position, i.e. every $d+1$ points are not on any possible hyperplane, and will discuss the case of arbitrary position afterwards.
Consider d points $P = \{p_1, p_2, \ldots, p_d\}$ out of S. These points define

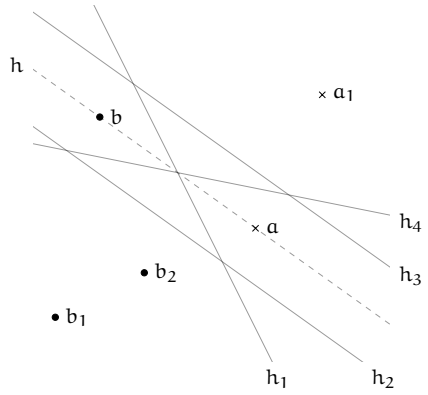


Figure 13: Example of the hyperplanes created for one pair

a hyperplane in $\mathbb{R}^d$, so regardless which inequality sign we would choose for the hyperplane, all $S \setminus P$ are divided into the same two sets S_1, S_2. Now for every ordered pair of disjoint sets $P_1 \cup P_2 = P$ we introduce a hyperplane, such that the hyperplane will additionally separate S_1 from P_2, S_2 from P_1 and P_1 from P_2. This is not a classical partition of P in two subsets, due to the fact that P_1 and P_2 are explicitly allowed to be empty, and we distinguish the order. For these d points we introduce $\sum_{i=0}^{d} \binom{d}{i} = 2^d$ hyperplanes in total, see Figure 13 for a two-dimensional example. The set Q is the conjunction of all these hyperplanes for every pair (or d-tuple), such that $|Q| = \binom{n}{d} 2^d$. Note that there are linear constraints that end up separating the same sets, thus Q is not minimal.
When S is not in general position, the sets S_1 and S_2 are not necessarily fixed, since more than d points could lay on the hyperplane. For the points on the hyperplane that are not in P, we could just compute in which set they belong when choosing the sets P_1 and P_2. This would not change the amount of linear constraints in Q.

Theorem 35.
Let S be a set of points in $\mathbb{R}^d$ with $|S| = n > d$. For a subset $A \subset S$ there exists a hyperplane in Q that separates A from $S \setminus A$ if and only if there exists a hyperplane that separates A from $S \setminus A$.

Proof. The if direction is trivial. Assume that there exists a hyperplane that separates A from $S \setminus A$. Now let P be the set of points that are on this hyperplane. If there are none, we can shift the hyperplane towards any point in the samples until at least one point is on the hyperplane. Unless $|P| < d$ we can rotate around these points until the changed hyperplane touches another point in S, until d or more points are on the hyperplane. So every point $S \setminus P$ is separated by the hyperplane defined by P. Despite the set A or $S \setminus A$ each point of P must be separated, since we took a hyperplane in Q that separates P in exactly that way. □

Example 10.
In Figure 13 the hyperplanes $h_1, \ldots, h_4$ *represent all hyperplanes that can be created from* $P = \{a, b\}$ *in* $\mathbb{R}^2$*, with* $S = \{a, a_1, b, b_1, b_2\}$*. The hyperplane* h *is induced by* P*, therefore divides* $S \setminus P$ *into* $S_1 = \{a_1\}$ *and* $S_2 = \{b_1, b_2\}$*. Every hyperplane* $h_1, \ldots, h_4$ *still separates* S_1 *from* S_2*. The hyperplane* h_1 *for example represents the choice of* $P_1 = \{a\}$ *and* $P_2 = \{b\}$*.*

Now consider the basic algorithm that will test every combination of k hyperplanes out of Q, these are $\binom{|Q|}{k}$ combinations. This leads to a total number of $\mathcal{O}(n^{kd}2^{kd})$ tests, which is polynomial for the number of sample points n for fixed dimension d and a fixed number of k hyperplanes.
For the PAC learning results we consider the following basic algorithm. As an input the algorithm takes (ε, δ, k), while the parameters d, k, f are arbitrary but fixed. Then the algorithm computes a number of needed samples $n(\varepsilon, \delta)$, and get the sample set S of size n. Now the basic algorithm will test for every $k' \in \{1, \ldots, k\}$ if there exist k' hyperplanes out of Q, such that no single sample in S is misclassified. If this is the case we terminate with a $k' \leqslant k$, or otherwise with a hypothesis using k hyperplanes with a minimal number of misclassified points in S. This leads to an overall running time of $\mathcal{O}(k(n(\varepsilon, \delta)^{kd}2^{kd}))$. When $n(\varepsilon, \delta)$ is a polynomial in $k, d, 1/\varepsilon$ and $1/\delta$, the running time is still polynomial in $1/\varepsilon$ and $1/\delta$. Now we will

Algorithm 4 : Basic Algorithm for PAC learning

Input : (ε, δ, k)
The parameters d, k and f are arbitrary but fixed.
Output : A linear arrangement used as a classifier, such that parts of the partition are mapped to a class.
begin
  Compute the number of needed samples $n(\varepsilon, \delta)$, and get the sample set S of size n.
  for $k' \in \{1, \ldots, k\}$ **do**
    Compute $Q' \subset Q$ with $|Q'| = k'$, such that the number t of misclassified samples in S is minimal.
    if $t = 0$ **then**
      return k' and the classifier created by Q' and S.
    else
      if $k' = k$ **then**
        return k and the classifier created by Q' and S.

first prove that this is a PAC learning algorithm and will improve the idea by changing the brute-force search through all possibilities into a more reasonable search due to column generation in Section 14.4.

14.3 THEORETICAL RESULTS

In this section we will present the main theoretical results for the basic algorithm 14.2.
As already mentioned similar results for the concept class are already known by [12]. The techniques and the algorithm used in this paper are different from ours, and as mentioned before were not known by us when we proved these statements.
In the first part we focus on PAC learning, while afterwards we will discuss the algorithm in the context of agnostic PAC learning. We will introduce our proof-technique, which is quite similar to the one used to prove the rectangle game (see Section 11.1.1.1) in the simple case of $\mathcal{C}_1^2$. Then we give the more technical proof for the concept class $\mathcal{C}_k^d$. After this we will use some results based on the VC-dimension to prove that our basic algorithm is also an efficient agnostic PAC learning algorithm.

14.3.1 *Probably Approximately Correct Learning*

In this section we will first prove that the basic algorithm is a PAC learning algorithm for the concept class $\mathcal{C}_1^2$. This is done for the benefit of a better understanding of the proof technique. Afterwards we will give the proof of $\mathcal{C}_k^d$, which obviously includes the previous statement but is much more technical.

Theorem 36.
The basic algorithm (Algorithm 4) is a PAC *learning algorithm for the concept class* $\mathcal{C}_1^2$.

Proof. Consider an arbitrary but fixed distribution $\mathcal{D}$ over $\mathbb{R}^2$ and values $\varepsilon \in (0,1)$, $\delta \in (0,1/2)$. There have to be sample points of each class, otherwise nothing has to be separated. In the general case, the error of a hypothesis hyperplane h to the target hyperplane h^* is defined by a double-cone T', see Figure 14 for an example. The degenerated case where both hyperplanes are parallel is discussed below.
This double-cone is well defined by the regions $not(h^*) \cap h$ and $h^* \cap not(h)$, since the orientation of both hyperplanes is fixed by the sample points. Additionally there is an intersection of h and h^* at the point x. We define a double-cone T, between the target hyperplane h and a hyperplane h', with an intersection in x, such that

$$Pr_{s\in\mathcal{D}X}[s \in T] = \varepsilon$$

and there exists an intersection of T and T' for any ε, which fixates the rotation around x, when $T' \neq \emptyset$. This is well defined, since at the start with $h' = h^*$ the probability is zero, and by rotating h' in point x this leads to probability one, when it fulfills $h' = h^*$ again. In case that $T' = \emptyset$, the probability of $Pr_{s\in\mathcal{D}X}[s \in T'] = 0$. The weight of T'

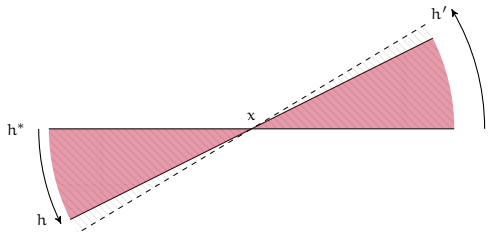


Figure 14: The error of the hypothesis hyperplane

has lesser weight than ε under $\mathcal{D}$ if and only if $T' \subseteq T$. This follows directly from the construction of T. Furthermore it holds that if a point in T appears in the sample S, then $T' \subseteq T$, since our algorithm separates the sample without error. The direct consequence is that the following conditional probability is 1.

$$Pr_{\mathcal{D}}[error(h) \leqslant \varepsilon | T \cap S \neq \emptyset] = 1$$

Now let us take a look at the probability that no point in T appears in S, denote $n = |S|$.

$$\begin{aligned} Pr_{\mathcal{D}}[T \cap S = \emptyset] &= Pr_{s_i \in_{\mathcal{D}} X}[s_1, \ldots, s_n \notin T] \\ &= (Pr_{s \in_{\mathcal{D}} X}[s \notin T])^n \\ &= (1 - Pr_{s \in_{\mathcal{D}} X}[s \in T])^n \\ &= (1-\varepsilon)^n \end{aligned}$$

If we choose n such that $(1-\varepsilon)^n \leqslant \delta$, then the probability that no point in T appears in S is greater than δ. Then the probability that at least one point appears in T, is at least $1-\delta$.
Now we are able to calculate the following probability.

$$\begin{aligned} Pr_{\mathcal{D}}[error(h) \leqslant \varepsilon] = \quad & Pr[error(h) \leqslant \varepsilon | T \cap S = \emptyset] Pr[T \cap S = \emptyset] \\ & + Pr[error(h) \leqslant \varepsilon | T \cap S \neq \emptyset] Pr[T \cap S \neq \emptyset] \\ \geqslant \quad & Pr[error(h) \leqslant \varepsilon | T \cap S \neq \emptyset] Pr[T \cap S \neq \emptyset] \\ \geqslant \quad & 1(1-\delta) \\ = \quad & (1-\delta) \end{aligned}$$

This leads to a lower bound of n,

$$n \geqslant (1/\varepsilon)\ln(1/\delta)\ .$$

Now we discuss the case where the hyperplanes are parallel. There are again two cases, but each of them is treated similarly. The first

case is where h and h^* are orientated in the same direction, therefore the true error T' is between these hyperplanes. The second case is that their orientation is not equal, then the error T' is defined by the complement of the region between these hyperplanes. We will take a closer look at the first case. We define T by a hyperplane h' also parallel to h^* and h, with the same orientation as h. In the first case T is defined by the region between h' and h^*. Now we move h' towards h until the point where the probability of T under $\mathcal{D}$ is equal to ε. This is only possible if the probability-mass on the negative side of h^* is greater than ε. If this were not the case, the true error is lesser than ε. As in the case of general position, we can conclude that the weight of T' exceeds weight ε under $\mathcal{D}$ if and only if $T' \supseteq T$. And furthermore that if $T' \supseteq T$ then no point in T appears in the sample S, by the same argument as above. The same is true for the second case, where only T' and T are defined by the complement. In this case we have to technically divide the region T into two parts. There is a contradiction if there is a point of the sample on the side of h^* where h is not, since then we would make an error in the sample.

□

Theorem 37.
The basic algorithm (Algorithm 4) is a PAC *learning algorithm for the concept class* $\mathcal{C}_k^d$.

Proof. Consider an arbitrary but fixed distribution $\mathcal{D}$ over $\mathbb{R}^d$ and values $\varepsilon \in (0,1)$, $\delta \in (0,1/2)$, and $k \in \mathbb{N}$. Since the algorithm would find a solution with no test error at least with the use of k linear constraints, this leads to an upper bound of $2k$ linear constraints in our discussion. We will name the k linear constraints of the target class $c_1, \ldots, c_k$, and the k linear constraints of the hypothesis $h_1, \ldots, h_k$. All those are later used as halfspaces, so each of them will represent a halfspace in $\mathbb{R}^d$. We define an index set $I = \{1, \ldots, k\}$, and a set R of all regions defined by all of the $2k$ hyperplanes. Proposition 33 states that the maximal number of regions defined by $2k$ hyperplanes is

$$|R| \leqslant 1 + 2k + \binom{2k}{2} + \cdots + \binom{2k}{d} . \tag{54}$$

Now let $R' \subset R$ be the subset of regions, where the class predicted by the target and the hypothesis are different. The general case of the *true* error, can then be specified by R'

$$\text{error}(h) = \sum_{r \in R'} \Pr_{s \in_{\mathcal{D}} X}[s \in r] .$$

For readability we write $\Pr[r]$ instead of $\Pr_{s \in_{\mathcal{D}} X}[s \in r]$ in the rest of the discussion.
Now we want to give a sketch of what we are going to do. The proof will be similar to the previous one for $\mathcal{C}_1^2$. In this proof we had constructed two double-cones T that had exactly the probability mass ε,

and T' which was the double-cone describing the difference between h and the target hyperplane h^*. Then we concluded that $T' \subseteq T$ if and only if T' has lesser probability-weight than ε. This concept will be used in this proof as well, with the difference that we do not have one region but $|R'|$ regions that sum up to the true error. Therefore when constructing the regions T_r we only let each of them have at most the probability-mass $\varepsilon/|R|$. We will then conclude that a similar statement like the previous if-only-if statement is true in this case. With this at hand we are able to estimate that the probability that the true error is smaller than ε is at least $1 - \delta$.

We take a look at a random region $r \in R'$, this region is defined by an intersection of $2k$ hyperplanes (or their negation). For readability we assume that r is the intersection of all $2k$ hyperplanes, without negation. The case with partial negated hyperplanes or less hyperplanes in the hypothesis is analogous. Now we construct T_r corresponding to the region r, while the following property must be fulfilled,

$$\exists p \in T_r \setminus r \implies r \subset T_r \,. \tag{55}$$

This is essential to conclude that

$$r \subset T_r \Leftrightarrow Pr[r] \leqslant Pr[T_r] \leqslant \varepsilon/|R|$$

holds.
If there are points in T_r that are not in r, then r is already a subset of T_r. First we define $l = 1$, $I' = I$, and $T_r^0 = \emptyset$. Then we take an arbitrary index $i \in I'$. For the chosen hyperplane h_i we create a hyperplane $h_i' = \neg c_1$, update $I' := I' \setminus i$, and define T_r^l.

$$T_r^l = \left(\bigcap_{u \in I} c_u \cap \bigcap_{v \in I'} h_v \cap h_i' \cup T_r^{l-1} \right) \setminus \bigcup_{\tilde{r} \in R \setminus R'} \tilde{r} \,. \tag{56}$$

At the beginning the set T_r^l is equal to T_r^{l-1} or more explicit the probability mass for T_r^l is equal to T_r^{l-1}.
Then we distinguish two cases of the hyperplanes h_i and c_1, either the hyperplanes are parallel or there exists an intersection x_i, this intersection is $(d-2)$–dimensional. In case of parallel hyperplanes we will edit the hyperplane h_i' by shifting it away from c_1 always staying parallel to each other. The direction is determined such that an infinitesimal shift leads to an intersection of the interiors of r and T_r^l. This will be continued unless, either the probability mass of T_r^l under $\mathcal{D}$ reaches exactly $\varepsilon/|R|$, or we shift to infinity and the mass is still lesser than $\varepsilon/|R|$. In case of an intersection we will rotate the hyperplane h_i' around the intersection x_i in the direction such that an infinitesimal rotation leads to an intersection of the interiors of r and T_r^l. Notice that at the beginning there is a intersection of r and T_r^l, but the probability mass is zero. We will continue to rotate unless

one of two things happens, either the probability mass of T_r^l under $\mathcal{D}$ reaches exactly $\varepsilon/|R|$, or h_i' is equal to c_1. By the fact that the integral over a continuous density function defines a continuous function, the intermediate value theorem states that a hyperplane that leads to a probability with exact $\varepsilon/|R|$ must exist, if the probability with $h_i' = c_i$ is greater than $\varepsilon/|R|$. If we stopped because $Pr[T_r] = \varepsilon/|R|$, we have finished the construction and set $T_r = T_r^l$. Otherwise we will repeat this procedure, set $l := l + 1$, and take again an arbitrary index $i \in I'$ and so on.
For T_r^1 the property (55) holds at the beginning, since $T_r^1 = \emptyset$. Regardless of whether the hyperplanes h_i and c_1 are parallel or not, $T_r^1 \subset r$ until the point where $h_i' = h_i$. At this point the sets T_r^1 and r are equal. After that it is possible there exist points, but it is not necessary. For any case $l > 1$, the set r is always a subset of T_r^l, since $r \subset T_r^1$ when there was no termination in the first iteration.
This algorithm defines T_r and has two possible outcomes. Either $Pr[T_r] = \varepsilon/|R|$, or $T_r = \bigcap_{u \in I} c_u \setminus \bigcup_{\tilde{r} \in R \setminus R'} \tilde{r}$ and $Pr[T_r] < \varepsilon/|R|$.

Algorithm 5 : Theoretical algorithm to construct T_r

Output : T_r: a convex region defined by hyperplanes, with property (55)

There are two possible outcomes, either $T_r = \varepsilon/|R|$, or $T_r = \bigcap_{u \in I} c_u \setminus \bigcup_{\tilde{r} \in R \setminus R'} \tilde{r}$ and $Pr[T_r] < \varepsilon/|R|$.

begin
- $T_r^0 = \emptyset$;
- $I' = I$;
- $l = 1$;
- **for** $i \in I'$ **do**
  - $T_r^l = \left(\bigcap_{u \in I} c_u \cap \bigcap_{v \in I'} h_v \cap h_i' \cup T_r^{l-1}\right) \setminus \bigcup_{\tilde{r} \in R \setminus R'} \tilde{r}$;
  - $h_i' = c_i$;
  - $I' = I' \setminus i$;
  - **repeat**
    - **if** h_i *and* c_1 *are not parallel* **then**
      - $x_i = h_i \cap c_i$;
      - rotate h_i' around x_i;
    - **else**
      - shift h_i' parallel to c_1
    - updating T_r^l;
  - **until** $h_i' \neq c_i$ *AND* $T_r^l \neq \varepsilon/|R|$;
  - **if** $T_r^l = \varepsilon/|R|$ **then**
    - **return** T_r^l;
  - $l = l + 1$;
- **return** T_r^l;

It now holds that

$$r \subset T_r \Leftrightarrow Pr[r] \leqslant Pr[T_r] \leqslant \varepsilon/|R| \,.$$

The if-part is trivial. The only-if-part follows from property (55) of T_r. Additionally it holds that if there exist a point $p \in T_r \cap S$, then $r \subset T_r$. Consider a point $p \in T_r \cap S$, then this point must be classified correctly, since the algorithm makes no sample error and no point in T_r is classified correctly. This leads to the fact that $p \in T_r \setminus r$, and since the property (55) holds, this leads to $r \subset T_r$.
First we estimate the following probability

$$Pr_S\left[Pr\left[r\right] > \frac{\varepsilon}{|R|}\right] = 1 - Pr\left[Pr\left[r\right] \leqslant \frac{\varepsilon}{|R|}\right]$$

The first probability is over the sample S, i.e. over the n drawings $s_1, \ldots, s_n \in_{\mathcal{D}} X$.

$$\leqslant 1 - Pr[T_r \cap S \neq \emptyset]$$

Because the event $T_r \cap S$ implies the event that $Pr[r] \leqslant \varepsilon/|R|$.

$$\begin{aligned}
&= 1 - (1 - Pr[T_r \cap S = \emptyset]) \\
&= Pr[T_r \cap S = \emptyset] \\
&\leqslant (Pr[s \notin T_r])^n \\
&= (1 - Pr[s \in T_r])^n \\
&= \left(1 - \frac{\varepsilon}{|R|}\right)^n
\end{aligned}$$

Now we take a look at the error of the hypothesis

$$\begin{aligned}
Pr_S\left[error(h) < \varepsilon\right] &\geqslant Pr\left[\bigcap_{r \in R'} Pr\left[r\right] \leqslant \frac{\varepsilon}{|R|}\right] \\
&= 1 - Pr\left[\bigcup_{r \in R'} Pr\left[r\right] > \frac{\varepsilon}{|R|}\right] \\
&\geqslant 1 - \sum_{r \in R'} Pr\left[Pr\left[r\right] > \frac{\varepsilon}{|R|}\right] \\
&\geqslant 1 - \sum_{r \in R'} \left(1 - \frac{\varepsilon}{|R|}\right)^n \\
&= 1 - |R'| \left(1 - \frac{\varepsilon}{|R|}\right)^n \\
&\geqslant 1 - |R| \left(1 - \frac{\varepsilon}{|R|}\right)^n
\end{aligned}$$

Now choose n such that $\delta \geqslant |R| \, (1 - \varepsilon/|R|)^n$.

$$\geqslant 1 - \delta$$

With the usage of the common inequality

$$(1-x) \leqslant e^{-x}$$

this leads to a lower bound of n,

$$n \geqslant \frac{|R|}{\varepsilon} \ln \frac{|R|}{\delta} .$$

This bound can be tightened, since the case where all regions are in R' is not possible. There is at least one region where correct classified points are, since all samples are correctly classified. With that we can reduce the bound from n by one, i.e.

$$n \geqslant \frac{|R|-1}{\varepsilon} \ln \frac{|R|-1}{\delta} .$$

□

14.3.2 *Agnostic Probably Approximately Correct Learning*

Agnostic PAC learning is based on the notion of VC-dimensions, and as we introduced in Section 11.1.4 it is essential that the VC-dimension of a hypothesis class must be finite. Therefore we have to argue that $\mathcal{C}_k^d$ has a finite VC-dimension.
Whenever a concept class C is of finite VC-dimension, then C_l, the set of all Boolean combinations formed from at most l concepts in C, is also of finite VC-dimension [23]. This leads to the fact that every convex l-gon is of finite VC-dimension, since it is formed by l-fold intersections of half-spaces (over $\mathbb{R}^d$), which are known to have the finite VC-dimension $d+1$. The hypothesis class $\mathcal{C}_k^2$ can also be defined over a function that maps convex regions describable by k fixed hyperplanes to a class.
Since the number of these convex regions are finite, we can formulate every concept of $\mathcal{C}_k^2$ as a Boolean combination formed over a finite number of k-gons, and thus its VC-dimension is finite, too. For a higher number of classes ($f \geqslant 2$) the VC-dimension is bounded by the VC-dimension of $\mathcal{C}_k^2$, since $\mathcal{C}_k^2 \subset \mathcal{C}_k^d$, for $d \geqslant 2$.
According to [31] an algorithm is an agnostic PAC-learner, if it misclassifies a minimal number of samples in S and its concept class has a finite VC-dimension. To add the missing statement, we have to conclude the following theorem.

Theorem 38.
Our basic algorithm computes for any $d, k \in \mathbb{N}$ *and any given samples* $S = \bigcup_{i=0}^{f} S_i$ *with* $|S| = n$ *a linear arrangement from* $\mathcal{C}_k^d$ *that misclassifies a minimal number of samples in* S.

This is a trivial statement for the basic algorithm, since it is formulated as a minimization problem that optimizes the number of misclassified samples.

Theorem 39.
The basic algorithm is an agnostic PAC *learning algorithm with the hypothesis class* $\mathcal{C}_k^d$.

Proof. The VC-dimension of $\mathcal{C}_k^d$ is finite for any fixed k. According to [31], this and the fact stated in Theorem 38 directly implies that our basic algorithm is an agnostic PAC learning algorithm. □

14.4 THE ALGORITHM BASED ON COLUMN GENERATION

In this section we describe the algorithm that was implemented to avoid the brute-force enumeration of all hyperplanes in Q, and therefore enlarge the reachability for higher parameters d and k. The algorithm CG computes for given $d, k \in \mathbb{N}$ and given n samples $S \subset \mathbb{R}^d \times \{1, \dots, f\}$ a function in $\mathcal{C}_k^d$, such that the number of misclassified points, and the used number of constraints are minimal with respect to $\mathcal{C}_k^d$. Additionally we can give the algorithm an input p, which specifies the relative learning accuracy as a termination criteria. By default p is set to 1, i.e. the algorithm will only terminate if there is no misclassified sample.
As mentioned before the hypothesis class is a function from the regions of the linear arrangements to the classes. The computation of the hypothesis in CG is done by solving an ILP with a column generation approach.
We will first introduce the ILP, and then state how we can incorporate the column generation approach to solve it. For every sample $s \in S$ we introduce a binary variable x_s that represents if the point s can be correctly classified, where $x_s = 1$ indicates an error and $x_s = 0$ indicates a correct classification. Further we introduce binary variables y_h that each represent a hyperplane h, where $y_h = 1$ indicates that h is used in the classifier, while $y_h = 0$ indicates it is not used. The ILP is then built by the objective function $c = \sum x_s$, and two types of constraints. The first type of constraint is a single constraint which fixes the number of used hyperplanes to k. The second type of constraint exists for every pair of points of *different* sets and basically stand for the ability that this pair is separable by the chosen y or not.

$$
\begin{aligned}
\min \quad & \sum x_s \\
\text{s.t.} \quad & \sum_{y_h \in Q} y_h = k \\
& x_s + x_t \geqslant 1 - \sum_{y_h \in Q \text{ separates } s\,t} y_h \qquad (57) \\
& x_s \in \{0, 1\} \\
& y_h \in \{0, 1\}
\end{aligned}
$$

The set Q is the finite subset of all hyperplanes in $\mathbb{R}^d$, described in Section 14.2.
This ILP (57) has a large amount of variables, since the number of hyperplanes in Q creates a massive amount of binary variables. To handle the large number of variables in Q we use a column generation approach, where only a small amount of variables will be present in the so-called RMP. Variables that can improve the current solution are found by the pricing problem and then added to the RMP. Therefore we introduce an RMP, where only a subset $Q' \subset Q$ of hyperplane-variables y are present.

$$\begin{aligned}
\min \quad & \sum x_s \\
\text{s.t.} \quad & \sum_{y_h \in Q'} y_h = k && (58) \\
& x_s + x_t \geqslant 1 - \sum_{y_h \in Q' \text{ separates } st} y_h && (59) \\
& x_s \in \{0,1\} \\
& y_h \in \{0,1\}
\end{aligned}$$

Typically we will solve this in a Branch and Price framework. Beginning in the root node, with k randomly chosen hyperplanes in the set Q', we solve the LP relaxation to optimality. Then the pricing problem is solved to find new variables $y \notin Q'$, such that y can improve the solution of the LP relaxation. We have different possibilities to terminate.

- The Branch-and-Price is solved to optimality.
- The global lower bound (dual bound), i.e. the worst lower bound of all nodes, implies an error that contradicts a relative learning accuracy of p.
- The best primal solution leads to an learning accuracy greater than p.
- At any time, we can terminate and output a heuristic solution.

14.4.1 *The Pricing Problem*

The goal of the pricing problem is to compute new variables, in our case hyperplanes that can help to improve the solution of the LP relaxation. In LP theory these variables are determined by having negative reduced costs. More specific the original pricing problem is in our case

$$c^* := \min_{y_h \in J \setminus Q'} \{-\pi^T a_{y_h}\}, \tag{60}$$

where π are the dual-factors of the optimal solution previously solved, and a_y are the factors of the newly found variable in the constraints

of the RMP. Thus every hyperplane y_h has a factor 1 for the constraint (58), since this constraint takes the sum over every $y \in Q'$. Thus the hyperplane y_h leads to a factor $(a_{y_h})_i = 1$ in the constraints of (59), if y_h separates the points corresponding to the constraint i, and 0 otherwise.
In most iterations it is enough to find a hyperplane such that $-\pi^T a_{y_h}$ is negative, since this would lead to a better solution of the RMP. But when the improvement is getting too small it is usually worth solving this problem exactly, i.e. to achieve lower bounds for the branch-and-bound tree. We set two parameters for the pricing, the maximal number of heuristic and exact computations for each node. Thus we compute the heuristic pricing as long as it is solvable and the maximal number of heuristic rounds is not reached. When the heuristic pricing is not able to compute a new variable with reduced costs, the exact pricing is triggered as long as the number of exact rounds already reached the maximal number of exact pricing. Note that either of the maximal values can be set to zero, while if both are set to zero no normal pricing would be computed. This problem without weights for every pair is known. In two dimensions this is called the stabbing line problem. In [5] an algorithm is introduced that solves the analogous unweighted problem in arbitrary dimension. Their algorithm finds a hyperplane stabber for n segments in $\mathbb{R}^d$ in $O(n^d)$. Since the unweighted problem can be seen as a subproblem of the weighted problem, this indicates that this problem is at least that hard. In our case the number of segments are the number of pairs that need to be separated, which is in $O(m^2)$.

14.4.1.1 *Heuristic Algorithm for the Pricing Problem*

This algorithm is a basic greedy approach for the pricing problem. The dual factors π give an order of the constraints and therefore we have an order of pairs that have to be separated. We first order the pairs according to their value, and then build up a set of pairs that is simultaneously separable by one hyperplane.

$$\begin{aligned} \max \quad & \varepsilon \\ \text{s.t.} \quad & a^T s_i \leqslant b - \varepsilon \quad \forall i \\ & a^T t_i \geqslant b + \varepsilon \quad \forall i \\ & -1 \leqslant a \leqslant 1 \\ & 0 \leqslant b, \varepsilon \end{aligned} \tag{61}$$

This LP formulation predefines on which side of the hyperplane each point will be. Because of this structure we always check if a point that is added to the LP is already used and therefore does not have to be added to the problem, and more importantly the side of the other point is already defined by the first choice. If a pair occurs where

both points are already used in the LP and are on the same side, it will not be separated. Since we first order the pairs according their value this leads to a simple greedy heuristic. To speed up the process we double the number of pairs that are inserted into the set after each successful insertion. An insertion is not successful and thus the pairs are not inserted, when the pairs are not separable simultaneously. At last we compute a linear constraint that separates at least the pairs that were in the last successful insertion. It can happen that more pairs are separated, but to compute the actual representation of this linear constraint we have to check every pair if it is separated by this constraint or not.

14.4.1.2 *Exact Algorithm for the Pricing Problem*

This algorithm is a slow but exact algorithm for the pricing problem. In this case we fall back to our basic algorithm, where we compute the reduced costs for every hyperplane for our finite subset Q. In fact we can speed up the process a little bit. In Section 14.2 we introduced the sets P,S_1,S_2,P_1 and $P_2 = P \setminus P_1$. We explained that we have to add 2^d hyperplanes in Q (for each subset of P). The reduced costs of each of these hyperplanes is given by the dual factors of the linear constraints, where the separated points are in S_1 and S_2, and additionally we have the factors that are produced by the actual subset P_1. We can therefore give a lower bound for the reduced costs for all hyperplanes defined by P. This is computed by adding the reduced costs by S_1 and S_2, for each point $p \in P$ the smaller of the two values corresponding to S_1 and S_2, and at last we add every reduced cost that can occur in P itself. If the lower bound is greater than zero, all 2^d hyperplanes will not have negative reduced costs.
As in the heuristic solution we can stop early once we found a variable with reduced costs. Only when we found the global minimum and therefore checked every hyperplane in Q, we can perhaps tighten the lower bound by the reduced costs. In this case the new lower bound is the maximum of the previous lower bound and $lp_{objval} + (c^*(k' - fV))$, where lp_{objval} is the objective value of the current LP solution, and fV is the number of already fixed variables due to branching decisions.
We also implemented an exact pricing computation, for the situation where all but one linear constraint is fixed due to branching decisions. In this case we use our LP formulation, but instead of making a greedy decision on which side the sample will lie, we compute the whole tree. Note that we do not have to branch a node further, if it is not separable itself.

14.4.1.3 *Branching Decisions and Farkas' Pricing*

This is a typical decision for all problems that use a branch and bound framework for solving. Especially in the context of column generation there is often the opportunity to make branching decisions that are unique to the problem, and often rely on the so-called original variables instead of the variables that are currently in the formulation. Since our case is not derived in that way we do not have the opportunity to use original variables, but instead have only the x and y variables present in our formulation. We decided to do a priority branch on the x variables, so if there are fractional x variables we will prefer them over any fractional y variable. This decision is supported by the fact that the branch on a x variable will split the problem into two subproblems, where one is stating that a certain point cannot be misclassified, and the other problem can be postponed, since its lower bound is at least 1. If only x variables are used, we therefore end up in a deep-first search like branch and bound tree, where the subproblem restricts many points to be misclassified. This can end up in the situation that the LP relaxation of the node is not solvable anymore and we would need to make a so-called *Farkas pricing*.
Since the problem is not solvable, there is no dual solution that could be used to formulate the pricing problem. Therefore in the Farkas pricing we use Farkas values. These are the solutions of the Farkas' infeasibility proof, or the Farkas' Lemma stated as Theorem 9 in Section 2.2.4. Those values are then used to formulate the Farkas pricing problem which uses those values just as we used the dual values in the pricing problem. In general the pricing problem would change in the way that there would be no objective values taken into consideration while solving the Farkas pricing, but since that is already the case in our pricing problem this changes nothing in our formulation. What really makes a difference is that we need to do Farkas pricing until we can prove that no variable with reduced costs exists, or until the LP is solvable due to new added variables. This leads to the fact that we will not stop after the stated maximal number of exact rounds, but have to compute further iterations.
Finally if the Farkas pricing is not able to produce variables with reduced costs, we have a proof that the LP is not solvable and therefore the branch and bound node can be cut. If we are in the case that we dived into a deep-first search, where only x variables where used in the branching decision, this means that for this k' the problem is not solvable without error and we can terminate the process.
Now we are able to point out the difference when we would use y variables in the branching decisions. First of all, the two nodes that are created by this decision are both viable and no trivial lower bound can be obtained for each of them. We end up with one subproblem that fixates a single constraint, and therefore only has to choose $k' - 1$ different linear constraints, additionally the lower bound computed

in the following pricing problems is reduced. The other subproblem would only forbid a specific linear constraint, or more precisely a specific combination of separated pairs. But since the linear constraint is most likely not taken for separating all of those pairs there is a high chance that a similar linear constraint will separate the same relevant pairs while it differs in pairs that are not relevant since they are separated by other linear constraints. This leads to a cascade where we have to branch multiple y variables to finally forbid that the relevant pairs are separated by one linear constraint. All those nodes are quite similar but must be treated separately.

14.4.1.4 *Three Cycle Cuts*

In theory we developed a method that could improve the problem that arises when branching on y variables. Instead of branching on y variables we add for this a linear constraint to make the current fractional solution infeasible, without changing the feasible set based on the binary variables, this is a so-called *cut*. This specific cut is not always applicable and will not entirely eliminate the need of branching on y variables, but will tackle a certain situation that is common in all cases where the number of classes exceeds two. This can get out of hand, when there are multiple of these situations in one sample, since it would take a lot of effort to prove that, for example, two of these situations are not separable by three linear constraints. In the original formulation we have the following linear constraints,

$$\begin{aligned} x_1 + x_2 &\geqslant 1 - y_1 - y_2 \\ x_1 + x_3 &\geqslant 1 - y_2 - y_3 \\ x_2 + x_3 &\geqslant 1 - y_1 - y_3 \,. \end{aligned}$$

And we would end up adding the linear constraint

$$x_1 + x_2 + x_3 \geqslant 2 - y_1 - y_2 - y_3 \,.$$

This new constraint would make the current fractional solution infeasible. And if the points x_1, x_2 and x_3 are from different classes we know that those three pairs cannot be separated by a single linear constraint, so the constraint will not change the feasible set on the binary variables. To use those cuts we need to search for all fractional variables y and check if there are triples that are connected that way. We can think of two strategies that can be considered. The first one is to simply add those constraints to the problem, this has the disadvantage that it will add many constraints to the initial problem, which will slow down the solving process in times where these constraints are not relevant. Even if some of them are relevant we would most likely end up adding more constraints than needed to solve the ILP to optimality. The other strategy is to check for usable cuts after each

relaxation is computed. In our view this is the more viable strategy. We do have to check every time if there exists such a triple, but we only have to check all triples of fractional y variables, instead of all variables.
The main challenge is that these constraints need to be considered in the pricing problem. Since they are constraints, they have corresponding dual variables which contribute to the reduced costs. Additionally for each new variable that is added through the pricing process we have to insert the variable in every cut, where it separates one of the corresponding pairs.

14.4.1.5 *Support Vector Like Heuristic*

The main bottleneck is the number of samples in our computation, since the exact pricing still needs to compute the reduced costs for every hyperplane in our finite subset Q. Therefore we incorporate an idea of support vectors to a heuristic, where we search for a subset of points which are essential for the determination of the k' hyperplanes.
This is a primal heuristic, which will not solve the pricing problem explicitly, but will eventually find hyperplanes that have reduced costs. We take the best primal solution found so far, and compute a subset of the samples $S' \subseteq S$. In this subset S' we take every misclassified point of the solution, and add for each hyperplane the substantial learning samples that define this hyperplane so far, i.e. for each hyperplane we take the d closest points on each side. For this sample set, we start an instance of CG that will not use this heuristic itself and let it run for several seconds. Whether or not it terminates in this short time frame, we can use the computed lower bound, and we check the best primal solution found so far. If we want to achieve a certain percentage of correct classified training samples, we can use the lower bound to terminate the current process, since the lower bound on S' is valid for S. Additionally every hyperplane of the best primal solution so far will be checked for reduced costs.
We trigger this heuristic only once per primal solution and only if the objective value is better than a certain relative bound (in our experiments 10% of the size of the training set), since early primal solutions misclassify a lot of samples, and therefore the reduction from S to S' would be not efficient.

14.4.2 *Margin Optimization*

After the ILP 57 is solved by the Branch and Price framework, we have minimized the number of misclassified samples by k hyperplanes. To maximize the margin of each of the k hyperplanes, we take the subset of $P \subset S$ which is not separated by any of the remaining $k-1$ hyperplanes and formulate a similar LP (61) like the one we used for

testing if the pairs are separable simultaneously. The test for separation is clearly not necessary, but the objective function will maximize the margin.

14.5 RANDOMIZED INCREASING THE SIZE OF SAMPLES

Since the main bottleneck for all approaches is the number of samples and therefore the amount of linear constraints that must be considered, we used another technique to withhold the number of constraints to a small level. Therefore we build up the sample set and solve many easier problems, with respect to the number of points in the sample, instead of solving one large problem. There are different options to increase the size of samples, but we ended up using a guided randomized increase. We will start with at least one sample of each class in a partial sample set. Then we solve the problem on the partial sample set. If it is not possible to separate those, we know it is not possible to solve the problem on the whole sample set. The current solution for the partial sample set is then used to find a new sample to increase the partial set. We assign a weight to every sample, which is initialized by 1. If the point is wrongly classified by the current solution the weight will be multiplied by 2. Then we draw a random number over the total weight of every current misclassified sample, and add this sample to the partial set.
By this scheme we will increase the chance of choosing a sample that was not separated by chance. A similar scheme is known for other randomized algorithms, typically used for so-called Las-Vegas-algorithms. In these algorithms typically the complexity of the problem is reduced to a reasonable amount by reducing the samples, but the main difference is that they usually can pinpoint to a constant number of samples. Additionally the most crucial fact is that these algorithms typically do not have to verify lower bounds, but instead already know that there is a solution for this problem. In our case we have to evaluate for every k if it feasible or not, until we end up with the lowest k that has a feasible solution. Therefore we increase the number of samples instead of restarting every time with the same amount of samples.
There is a freedom about how many samples we can add in each iteration. Our testings showed that it is most viable if we only add one sample at a time, since in this case we can sometimes rearrange one linear constraint to separate the additional sample. Therefore we simply test if a neighboring of the misclassified point is appointed to the same class or to no class at all. Then we try to adapt the corresponding linear constraint that separates those neighbor regions. This is done by computing in a single LP if the points that are only separated by this constraint are still separable if we switch the side of the misclassified point.

The worst case runtime for this algorithm combined with the basic algorithm is obviously worse than the worst case runtime for the basic algorithm itself. This is based on the fact that there is no maximum number of samples that would guarantee that it would lead to an optimal solution. Therefore the worst case is to add all samples once at a time, and at the end solve the complete problem anyway. On the other hand the expected runtime should be much lower, but is not computed in this thesis.

14.5.1 *Computing Linear Constraints*

For both the basic and the ILP-approach, we need to compute all relevant linear constraints defined by the samples. Since most of the linear constraints where already computed in the previous iteration it is obvious that we want to use this information. The problem which occurs is that we only want to add each binary variable once. As mentioned before we can differentiate the linear constraints by the fact which pairs they separate. For the subproblem it is only relevant to differentiate by pairs that are present in the subproblem. Now we want to use the previous computation and do not go over all $\binom{n}{d}$ combinations again. Thus we have to save the linear constraints of the previous computation with respect to all points of the sample. Then we compute all new linear constraints which can be created by the new sample and $d-1$ other samples of the partial sample set. If they are unique we will save them together with the previous linear constraints for the next iteration. After that we evaluate which of those are unique under the partial sample, and take only those into consideration. This is a significant overhead we have to take for reusing the previously computed linear constraints, and it is not clear if this pays off.

15 EVALUATION

In this chapter we want to evaluate our algorithm described in Section 14.4. We will first compare our different approaches among another. First we will introduce the experimental setup, i.e. the different algorithmic methods and the dataset used for the evaluation. In the next section we will compare the different algorithmic choices with different sets of parameters, and make a final choice for the parameters and the overlying algorithm. Then we will take this candidate and will compare it with well-known machine learning algorithms, namely Support Vector Machine (SVM) and k-Nearest Neighbor (kNN). Finally we will give a conclusion of the evaluation.

15.1 EXPERIMENTAL SETUP

For benchmarks we use two different types of datasets. First we created datasets for ourself. Therefore we defined the dimension d, the number of linear constraints k, the number of classes f, and always chose the number of samples. We then randomly chose k linear constraints in dimension d, and mapped the regions of this linear arrangement to the f classes. After that we drew samples and mapped them to the corresponding class. In that way we created a set with 300 samples. We always made sure that every class was mapped to at least one region, but it can occur that fewer linear constraints are needed to separate those regions, then the ones used in the construction. This is more likely when we use fewer samples, but it can also occur when we would use an infinite amount of samples. The other datasets are a set of known and not artificial examples, where it is mostly unknown how many linear constraints are needed to separate the classes. Namely we used the datasets Wisconsin Diagnostic Breast Cancer [53], Iris [25] and Wine [1], all found in the UCI *Machine Learning Repository* [38].

WISCONSIN DIAGNOSTIC BREAST CANCER (WDBC) A well-known data set that contains 569 patterns with 30 real-valued input features for a binary classification task. This set is linearly separable. In 1992 [8] introduced a linear program that solved this problem. Therefore they used a projection in the 3-D space of Worst Area, Worst Smoothness and Mean Texture.

IRIS PLANTS DATABASE (IRIS) This is a well-known dataset with 3 classes, containing 50 patterns each, and is taken over 4 real-valued input features. This set in total is not linearly separa-

ble, but one class is linearly separable from the other two. It is known for very low misclassification rates.

WINE RECOGNITION (WINE) A dataset of 178 patterns over 3 classes, and is taken over 13 real-valued input features. Created in 1992, it was shown that it was separable by RDA functions, but is not linearly separable.

In all our comparisons we use a single 10-fold cross-validation, and timeout of 300s for each fold.

15.1.1 *Implementation*

We implemented the different algorithms in C++, where we used scip [41] as a branch-and-price framework for the CG-approach. All LPs and the ILP for the ILP-approach were solved with gurobi [30], and to solve systems of linear equations we used eigen [29].

15.2 EVALUATION OF THE DIFFERENT ALGORITHMS AND STRATEGIES

We will first take a look at the performance of the different algorithms, the basic algorithm (B), the ILP-approach (ILP) where all constraints that are defined by the samples are precomputed, and the CG-approach (CG). Further we will evaluate different strategies stated in Section 14.4. Those algorithms share some parameters, i.e. INC, ADAPT, while all other parameters are not applicable to all algorithms. Therefore CG is specified by the following input parameters: INC, ADAPT, WARMSTART, # EX.ROUNDS, # HEUR.ROUNDS, SPECIAL CASE, EXACT EARLY EXIT. While the algorithms ILP and B are specified by these input parameters: INC, ADAPT, WARMSTART. The parameters are, except for # EX.ROUNDS and # HEUR.ROUNDS, all binary parameters, and they describe the following:

METHOD: States the method which is used, 0 stands for the CG approach, 1 for the basic algorithm and 2 for the ILP-approach.

INC.: States if we use the randomized increase of the sample size.

ADAPT: States if we try to adapt the old solution by changing only one constraint.

WARMSTART: Whether to reuse the previous computed linear constraints.

BRANCH: If this is used we prioritize branching on x variables, otherwise we leave it to the solver to make the branching decision.

EX.ROUNDS: States the number of exact rounds the pricing will do for each node in the branch and bound tree.

HEUR.ROUNDS: The number of heuristic rounds the pricing will do for each node in the branch and bound tree.

SPECIAL CASE: Whether or not we will treat a search for the last hyperplane differently than other exact pricing steps.

For this evaluation we only used artificial produced datasets that are separable by linear constraints. In this evaluation we will only focus on the runtime of the algorithms, since the quality of the solution does not depend on the algorithmic choice, but mainly on the choice that was made in the randomized approach (INC), and the choice between all feasible solutions. Note that neither is driven by the algorithmic choice of the method. To improve readability we do not state every result for every parameter setting, but only give a good representation of different candidates and summarize the impacts of every parameter. First we can state that the CG had a timeout in 110 out of 180 instances, whenever the whole dataset is used, i.e. the parameter *inc* is set to false. Additionally the basic approach (B) had a timeout in every single instance, when the parameter *inc* was set to false.
We will now go through all of the parameters and discuss their impact on the performance of the algorithm.

METHOD: Here we can state as a general rule that the CG-approach and the ILP-approach are superior to the basic algorithm despite the parameters that were chosen. The comparison between the CG-approach and the ILP-approach is more delicate and we will discuss that afterwards.

INC.: For the ILP and the basic algorithm both algorithms were not able to start the optimization process, because the number of linear constraints that have to be computed is to large. Therefore all instances of these algorithms that did not use the randomized approach to increase the sample set produced a timeout.
For the CG-approach the impact is still major, but there are instances where the approach found a solution. Still there were 110 out of 180 instances where it produced a timeout.

WARMSTART: In the CG-approach this means that the linear constraints computed in the previous iteration are directly added to the next problem, the impact is mixed. While the number of timeouts increases with the warmstart, the total runtime decreases. This behavior could be explained by the fact that the use of a previous linear constraint can lead to a local minimum, which is time consuming to leave through exact pricing steps. In the ILP algorithm, this option increases both timeouts and the average runtime.

ADAPT: This parameter has a major impact on both algorithms. The runtime is reduced by more than 50% in both cases.

BRANCH: The branching decision implied no significant impact on the dataset.

EX.ROUNDS: Also the number of exact rounds had no further impact.

HEUR.ROUNDS: When we compare the runs where the number of heuristic rounds are 20 and 100, the relative improvement of the runtime with 100 is about 25%.

SPECIAL CASE: This option improves the runtime of the algorithm by about 8%.

Further it is notable that the increase of linear constraints and the number of classes where both manageable, while the dimension of 4 was enough to trigger timeouts, although both the linear constraints, and the number of classes were set to 2. This indicates that the dimension is the major factor in the computational complexity of the problem.
We can state for this dataset that the CG-approach is superior to the ILP-approach despite of the parameter choice as long as *inc* is set to true. The best performing parameter choice on this dataset is therefore to use *Inc*, *Adapt* and the *SpecialCase*, while the number of heuristic rounds is set to 100, and the number of exact rounds to 1. When we fixate all other parameters and only vary the number of heuristic rounds, the relative improvement from 20 to 100 rounds is reduced to 8%.

15.3 EVALUATION COMPARED TO RELATED METHODS

In the previous section we concluded that the CG-algorithm was superior to both the ILP and the basic-approach. Further we defined a choice of parameters for all flags based on the performance in the previous section. In this section we want to compare the CG-approach to the well-known procedures of support vector machines [13], with linear kernels, and k-Nearest Neighbor [26]. Both algorithms are available in Weka, a tool developed by [56]. We used Weka to run the datasets for both algorithms with different settings. Since both algorithms are well-known we will only give a very short introduction to both of them.
Support vector machines (SVM) had their breakthrough in 1992, where a implementation by Chang and Lin [13] was published. Since then it is a state-of-the-art classification method. It is known for its high accuracy, and the ability to deal with high-dimensional data. SVM is defined as a two-class classifier, therefore we used the multi-class version that is also implemented in libSVM [13]. The rough idea behind

SVM is to solve a non-linear optimization problem where the margin between a hyperplane and the samples is maximized. In the optimization the hyperplane can then be swapped to a kernel-function which allows the decision boundary to be non-linear. Therefore there are multiple parameters that need to be set, where the most important parameters are the choice of the kernel, and the penalization factor C in the objective function for points that are misclassified or inside the margin. A large C leads to fewer of these points, while a small C will focus on maximizing the margin.

k-*Nearest Neighbor* (KNN), first published in 1951 by [26], is a simple but popular method. Based on a distance measure, it maps a new sample to the class with the most members present in the k nearest neighbors, based on the chosen distance measure. While we will use only the euclidean distance, other distance measures are also viable choices. Usually k is set to a small uneven number to dodge the case where there are two classes with the same number of neighbors. This works at least in the case where the problem only consists of two classes.

For this evaluation we used all datasets described in Section 15.1, including the previously used artificial dataset that we constructed. In contrast to the previous section we will focus on the quality of the solutions, instead of the runtime. As a measure we will look at the percentage of correctly classified samples, combined with the kappa value. To describe the kappa value in a nutshell, this value combines the observed accuracy while taking into account the expected accuracy of a dataset. In other words, it will take the distribution of the classes into account. To categorize poor, good and excellent classifiers by the kappa value is not a trivial task. As an estimate we can consider kappas below 0.4 as poor, between 0.4 and 0.75 as good, and over 0.75 as excellent.

In Table 9 we therefore state the runtime in seconds, the percentage of correct classified samples (% correct), the kappa value, and for our algorithm the number of timeouts (# to).

We can observe that the runtime of our algorithm is clearly not yet practical for real datasets, since the dimension representing the number features is typically higher then 4, and as we saw in our artificial examples this leads to serious problems in the computational time. For the quality of the solution we observe that in the *WDBC*, *Iris* and the *Wine*-dataset our CG-approach leads to a higher classification error, but is still comparable. Its kappa value of all three datasets is above .87, which indicates that it is able to compute a reasonable classifier for those datasets. When we take a look at the artificial datasets we created, the CG-approach is able to outperform the other two candidates, when given enough time. There the SVM is not able to separate the different cells, while kNN is able to classify a significant amount.

Table 9: Comparison of the Runtime and Accuracy to Related Methods

	CG	SVM			kNN		
		C=0.1	C=1	C=10	K=1	K=2	K=3
WDBC							
runtime (in s)	3.01	0.71	2.65	4.00	0.01	0.01	0.01
% correct	94.03	94.73	95.44	95.08	95.96	95.96	96.84
Kappa	0.87	0.89	0.90	0.89	0.91	0.91	0.93
# to	0						
Iris							
runtime (in s)	138.36	0.01	0.00	0.00	0.00	0.00	0.00
% correct	92.00	96.67	96.67	96.00	95.33	94.67	95.33
Kappa	0.88	0.95	0.95	0.94	0.93	0.92	0.93
# to	2						
Wine							
runtime (in s)	3.86	0.35	0.57	1.42	0.00	0.00	0.00
% correct	94.35	95.49	95.52	96.08	94.97	95.56	94.97
Kappa	0.91	0.93	0.93	0.94	0.92	0.93	0.92
# to	0						
$A(1000, 5, 3, 2)$							
runtime (in s)	18.13	0.03	0.04	0.24	0.01	0.00	0.00
% correct	99.40	78.50	78.50	78.50	96.90	95.60	95.90
Kappa	0.99	0.67	0.67	0.67	0.95	0.93	0.94
# to	0						
$A(1000, 6, 3, 2)$							
runtime (in s)	187.67	0.03	0.07	0.35	0.01	0.00	0.00
% correct	97.90	73.70	73.50	73.50	94.00	93.00	93.40
Kappa	0.97	0.60	0.60	0.60	0.91	0.89	0.90
# to	1						
$A(300, 5, 3, 2)$							
runtime (in s)	26.40	0.02	0.02	0.07	0.00	0.00	0.00
% correct	96.67	68.00	68.33	68.00	88.33	84.67	88.00
Kappa	0.95	0.49	0.50	0.49	0.82	0.76	0.81
# to	0						
$A(300, 6, 3, 2)$							
runtime (in s)	217.38	0.01	0.01	0.04	0.00	0.00	0.00
% correct	92.33	77.33	77.33	77.33	91.00	89.67	91.33
Kappa	0.80	0.00	0.00	0.00	0.74	0.68	0.74
# to	2						

16 CONCLUSION

In this part we introduced a PAC learning algorithm for $\mathcal{C}_k^d$, see Section 14.3. While the worst case runtime of the algorithm is still polynomial for fixed d and k, the given bound is also the best case, which motivated us to improve the algorithm. Therefore we formulated an ILP and at last implemented a column generation (CG) approach to solve it. Further we introduced two main approaches to handle the number of samples. First we randomized an incremental increase of the sample size, such that we can start with a small sample size and the randomized algorithm chooses meaningful samples to build up a good representation of the dataset. This motivated a second approach, where instead of solving the CG-algorithm, we first search if there is a simple solution be changing only one linear constraint in the description - this was done by using a linear program.
In the evaluation we confirmed that each mentioned step improved the practical runtime of the algorithm significantly, such that the chosen candidate of Section 15.2 is now able to handle 1000 samples, while both the basic and the ILP-approach were not able to start to compute separation, since the precomputation of all relevant linear constraints was to time consuming. Still the comparison of more realistic datasets, and other often used machine learning algorithms showed several things. The runtime is still not practically usable. We are now able to handle more samples, but especially the dimension is a factor that cannot be handled well by the algorithm. Additionally we deem that the expressibility with a fixed k is insufficient to give a reasonable classifier in many cases.

16.1 FUTURE WORK

There are still many things that could be improved in the CG approach. The main problem here lies within the case of the exact pricing problem. If we could find a way to solve this in a more reasonable time frame, like in the special case, where only one linear constraint is searched, this would improve the overall computational time significantly. There is also a concept of combining column and row generation [44], which could be advantageous to our approach of handling both separately, with the column generation and the randomized algorithm which essentially reduces rows and variables in the ILP. As for the use of parallelization, this could improve the computation of the exact pricing, since we search for a global minimum of a set, where the computation of each element does not depend on the

outcome of another. Furthermore a CG approach has always several possibilities for improvement. As for an example we tried to use a primal heuristic for improving the lower bound earlier in the search tree. In our case this would reduce the time we spent searching for k' for which the samples are not separable. In the end we did not use it, due to the randomized approach which increases the sample size incrementally.
Another point for improvement would be a good starting heuristic for a useful set of linear constraints. This perhaps could be done by using a clustering algorithm and explicitly computing the linear constraints that describe the corresponding Voronoi cells.

BIBLIOGRAPHY

[1] Stefan Aeberhard, Danny Coomans, and Olivier de Vel. "The classification performance of RDA." In: *Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland, Tech. Rep* (1992), pp. 92–01.

[2] Aws Albarghouthi and Kenneth L McMillan. "Beautiful interpolants." In: *Computer Aided Verification*. Springer. 2013, pp. 313–329.

[3] Ernst Althaus, Björn Beber, Joschka Kupilas, and Christoph Scholl. "Improving interpolants for linear arithmetic." In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2015, pp. 48–63.

[4] Ernst Althaus, Björn Beber, Werner Damm, Stefan Disch, Willem Hagemann, Astrid Rakow, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. "Verification of linear hybrid systems with large discrete state spaces using counterexample-guided abstraction refinement." In: *Science of Computer Programming* (2017).

[5] David Avis and Mike Doskas. "Algorithms for high dimensional stabbing problems." In: *Discrete applied mathematics* 27.1-2 (1990), pp. 39–48.

[6] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[7] B. Becker, W. Damm, M. Fränzle, E.-R. Olderog, A. Podelski, and R. Wilhelm. *SFB/TR 14 AVACS – Automatic Verification and Analysis of Complex Systems*. 2004. URL: http://www.avacs.org.

[8] Kristin P Bennett and Olvi L Mangasarian. "Robust linear programming discrimination of two linearly inseparable sets." In: *Optimization methods and software* 1.1 (1992), pp. 23–34.

[9] Avrim Blum and Ronald L. Rivest. "Training a 3-Node Neural Network is NP-Complete." In: *Machine Learning: From Theory to Applications - Cooperative Research at Siemens and MIT*. 1993, pp. 9–28. DOI: 10.1007/3-540-56483-7_20. URL: http://dx.doi.org/10.1007/3-540-56483-7_20.

[10] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. "Learnability and the Vapnik-Chervonenkis dimension." In: *Journal of the ACM (JACM)* 36.4 (1989), pp. 929–965.

[11] Robert S. Boyer and J. Strother Moore. *Program verification*. Tech. rep. TEXAS UNIV AT AUSTIN INST FOR COMPUTING SCIENCE and COMPUTER APPLICATIONS, 1984.

[12] Nader H Bshouty, Sally A Goldman, H David Mathias, Subhash Suri, and Hisao Tamaki. "Noise-tolerant distribution-free learning of general geometric concepts." In: *Journal of the ACM (JACM)* 45.5 (1998), pp. 863–890.

[13] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines." In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 27:1–27:27.

[14] Stephen A Cook. "The complexity of theorem-proving procedures." In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158.

[15] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. "Verification of synchronous sequential machines based on symbolic execution." In: *International Conference on Computer Aided Verification*. Springer. 1989, pp. 365–373.

[16] William Craig. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory." English. In: *Journal of Symbolic Logic* 22.3 (1957), pp. 269–285. ISSN: 00224812.

[17] Robert J Dakin. "A tree-search algorithm for mixed integer programming problems." In: *The computer journal* 8.3 (1965), pp. 250–255.

[18] Werner Damm, Stefan Disch, Hardi Hungar, Swen Jacobs, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. "Exact State Set Representations in the Verification of Linear Hybrid Systems with Large Discrete State Space." In: *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*. Vol. 4762. LNCS. 2007, pp. 425–440.

[19] Werner Damm, Henning Dierks, Stefan Disch, Willem Hagemann, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. "Exact and Fully Symbolic Verification of Linear Hybrid Automata with Large Discrete State Spaces." In: *Science of Computer Programming* 77.10-11 (2012), pp. 1122–1150. ISSN: 0167-6423.

[20] George B Dantzig. "Programming in a linear structure." In: *Washington, DC* (1948).

[21] George B Dantzig. "Programming of interdependent activities: II mathematical model." In: *Econometrica, Journal of the Econometric Society* (1949), pp. 200–211.

[22] George B Dantzig. "Maximization of a linear function of variables subject to linear inequalities." In: *New York* (1951).

[23] Richard M Dudley. "A course on empirical processes." In: *Ecole d'été de Probabilités de Saint-Flour XII-1982*. Springer, 1984, pp. 1–142.

[24] H. Edelsbrunner. *Algorithms in combinatorial geometry*. Springer Science & Business Media, 2012.

[25] Ronald A Fisher. "The use of multiple measurements in taxonomic problems." In: *Annals of human genetics* 7.2 (1936), pp. 179–188.

[26] Evelyn Fix and Joseph L Hodges Jr. *Discriminatory analysis nonparametric discrimination: consistency properties*. Tech. rep. California Univ Berkeley, 1951.

[27] Goran Frehse. "PHAVer: Algorithmic Verification of Hybrid Systems past HyTech." In: *STTT* 10.3 (2008), pp. 263–279.

[28] Paul C Gilmore and Ralph E Gomory. "A linear programming approach to the cutting stock problem—Part II." In: *Operations research* 11.6 (1963), pp. 863–888.

[29] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: `http://eigen.tuxfamily.org`.

[30] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: `http://www.gurobi.com`.

[31] David Haussler. "Decision theoretic generalizations of the PAC model for neural net and other learning applications." In: *Information and computation* 100.1 (1992), pp. 78–150.

[32] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming." In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM. 1984, pp. 302–311.

[33] Richard M Karp. "Reducibility among combinatorial problems." In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[34] Michael J Kearns, Robert E Schapire, and Linda M Sellie. "Toward efficient agnostic learning." In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM. 1992.

[35] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[36] Leonid G Khachiyan. "A polynomial algorithm in linear programming." In: *Doklady Akademiia Nauk SSSR*. Vol. 244. 1979, pp. 1093–1096.

[37] Ailsa H Land and Alison G Doig. "An automatic method of solving discrete programming problems." In: *Econometrica: Journal of the Econometric Society* (1960), pp. 497–520.

[38] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: `http://archive.ics.uci.edu/ml`.

[39] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. "A comparative study of incremental constraint solving approaches in symbolic execution." In: *Haifa Verification Conference*. Springer. 2014, pp. 284–299.

[40] Rüdiger Loos and Volker Weispfenning. "Applying linear quantifier elimination." In: *The computer journal* 36.5 (1993), pp. 450–462.

[41] Stephen J. Maher et al. *The SCIP Optimization Suite 4.0*. eng. Tech. rep. 17-12. Takustr.7, 14195 Berlin: ZIB, 2017.

[42] K. L. McMillan. "An Interpolating Theorem Prover." In: *Theoretical Computer Science* 345.1 (2005), pp. 101 –121. ISSN: 0304-3975.

[43] Nimrod Megiddo. "On the complexity of polyhedral separability." In: *Discrete & Computational Geometry* 3.1 (1988), pp. 325–337.

[44] Ibrahim Muter, Ş İlker Birbil, and Kerem Bülbül. "Simultaneous column-and-row generation for large-scale linear programs with column-dependent-rows." In: *Mathematical Programming* 142.1-2 (2013), pp. 47–82.

[45] Greg Nelson and Derek C Oppen. "Simplification by cooperating decision procedures." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.2 (1979), pp. 245–257.

[46] Florian Pigorsch and Christoph Scholl. "Lemma Localization: A Practical Method for Downsizing SMT-Interpolants." In: *Proc. of DATE*. Grenoble, France, 2013, pp. 1405–1410. ISBN: 978-1-4503-2153-2.

[47] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. "Constraint Solving for Interpolation." In: *Proc. of VMCAI*. 2007.

[48] Christoph Scholl, Stefan Disch, Florian Pigorsch, and Stefan Kupferschmid. "Computing Optimized Representations for Nonconvex Polyhedra by Detection and Removal of Redundant Linear Constraints." In: *Proc. of TACAS*. 2009, pp. 383–397. ISBN: 978-3-642-00767-5.

[49] Christoph Scholl, Florian Pigorsch, Stefan Disch, and Ernst Althaus. "Simple interpolants for linear arithmetic." In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE. 2014, pp. 1–6.

[50] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[51] Robert E Shostak. "A practical decision procedure for arithmetic with function symbols." In: *Journal of the ACM (JACM)* 26.2 (1979), pp. 351–360.

[52] Richard P Stanley et al. "An introduction to hyperplane arrangements." In: *Geometric combinatorics* 13 (2004), pp. 389–496.

[53] W Nick Street, Olvi L Mangasarian, and William H Wolberg. "An inductive learning approach to prognostic prediction." In: *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 522–530.

[54] Leslie G Valiant. "A theory of the learnable." In: *Communications of the ACM* 27.11 (1984), pp. 1134–1142.

[55] VN Vapnik and A Ya Chervonenkis. "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities." In: *Theory of Probability & Its Applications* 16.2 (1971), pp. 264–280.

[56] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *The WEKA Workbench. Online Appendix for Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. San Francisco, CA, USA, 2016.