Binary Decision Diagrams and Integer Programming

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von

Markus Behle

Saarbrücken
2007

Datum des Kolloquiums:
    22. Dezember 2007
Dekan der Naturwissenschaftlich-Technischen Fakultät I:
    Prof. Dr. Thorsten Herfet
Vorsitzender:
    Prof. Dr. Gerhard Weikum, Max-Planck-Institut für Informatik, Saarbrücken
Gutachter:
    Prof. Dr. Friedrich Eisenbrand, Universität Paderborn
    Prof. Dr. Kurt Mehlhorn, Max-Planck-Institut für Informatik, Saarbrücken
Beisitzer:
    PD Dr. Ernst Althaus, Max-Planck-Institut für Informatik, Saarbrücken

Abstract

In this work we show how Binary Decision Diagrams can be used as a powerful tool for 0/1 Integer Programming and related polyhedral problems.

We develop an output-sensitive algorithm for building a threshold BDD, which represents the feasible 0/1 solutions of a linear constraint, and give a parallel *and*-operation for threshold BDDs to build the BDD for a 0/1 IP. In addition we construct a 0/1 IP for finding the optimal variable order and computing the variable ordering spectrum of a threshold BDD.

For the investigation of the polyhedral structure of a 0/1 IP we show how BDDs can be applied to count or enumerate all 0/1 vertices of the corresponding 0/1 polytope, enumerate its facets, and find an optimal solution or count or enumerate all optimal solutions to a linear objective function. Furthermore we developed the freely available tool `azove` which outperforms existing codes for the enumeration of 0/1 points.

Branch & Cut is today's state-of-the-art method to solve 0/1 IPs. We present a novel approach to generate valid inequalities for 0/1 IPs which is based on BDDs. We implemented our BDD based separation routine in a B&C framework. Our computational results show that our approach is well suited to solve small but hard 0/1 IPs.

Kurzzusammenfassung

In dieser Arbeit zeigen wir, wie Binary Decision Diagrams (BDDs) als ein mächtiges Werkzeug für die 0/1 Ganzzahlige Programmierung (0/1 IP) und zugehörige polyedrische Probleme eingesetzt werden können.

Wir entwickeln einen output-sensitiven Algorithmus zum Bauen eines Threshold BDDs, der die zulässigen 0/1 Lösungen einer linearen Ungleichung darstellt, und beschreiben eine parallele *und*-Operation für Threshold BDDs, um den BDD für ein 0/1 IP zu bauen. Des Weiteren konstruieren wir ein 0/1 IP zum Finden der optimalen Variablenordnung und zum Berechnen des Variablenordnung Spektrums eines Threshold BDDs.

Zur Untersuchung der polyedrischen Struktur eines 0/1 IPs zeigen wir, wie man mit Hilfe von BDDs alle 0/1 Ecken des dazugehörigen 0/1 Polytops zählt oder enumeriert, seine Facetten enumeriert und zu einer linearen Zielfunktion eine optimale Lösung findet oder alle optimalen Lösungen zählt oder enumeriert. Darüber hinaus haben wir das frei erhältliche Tool `azove` entwickelt, welches bestehende Codes für die Enumerierung von 0/1 Punkten geschwindigkeitsmäßig übertrifft.

Branch & Cut ist heutzutage die Methode der Wahl zum Lösen von 0/1 IPs. Wir beschreiben einen neuartigen Ansatz zur Generierung zulässiger Ungleichungen für 0/1 IPs, der auf BDDs basiert. Unsere BDD-basierte Separierungsroutine haben wir in einem B&C Framework implementiert. Unsere Rechenresultate zeigen, dass unser Ansatz gut zum Lösen kleiner und zugleich schwieriger 0/1 IPs geeignet ist.

Acknowledgments

Many people influenced my work on this thesis. First of all, I thank my advisor Prof. Dr. Fritz Eisenbrand for his advice, encouragement and guidance during the last years. I am thankful for giving me the opportunity to work in his former Discrete Optimization group at the Max-Planck-Institut für Informatik.

I was most fortunate to meet Prof. Dr. Bernd Becker and Ralf Wimmer from the Albert-Ludwigs-Universität Freiburg. They introduced me to the field of binary decision diagrams and aroused my interests in combining BDDs with integer programming. I want to thank them for the very fruitful collaboration during the last years.

Many thanks go to all people at the Max-Planck-Institut für Informatik for creating such a fruitful and lively environment. In particular, I am grateful to Andreas Karrenbauer for many interesting discussions (not only) concerning research related topics. Furthermore I thank Ernst Althaus and Stefan Funke for their interests in my work and spending time on discussions. In addition, I thank Ernst Althaus for reading parts of the manuscript and his helpful comments.

I am grateful to Prof. Dr. Kurt Mehlhorn for his instant commitment to become a referee of this thesis and for giving me the possibility to finish my work at the MPI.

Contents

1 Introduction

1.1 Motivation

Binary Decision Diagrams (BDDs for short) are a datastructure represented by a directed acyclic graph, which aims at a *compact* and *efficient* representation of boolean functions. Since their significant extension in 1986 in the famous paper by Bryant [Bry86] they have received a lot of attention in fields like computational logics and hardware verification. They are used as an industrial strength tool, e.g. in VLSI design [MT98].

One class of BDDs are the so-called *threshold BDDs*. A threshold BDD represents in a compact way the set of 0/1 vectors which are feasible for a given linear constraint. As there is an obvious relation to the Knapsack problem, and thus to 0/1 integer programming in general, we were attracted by this class of BDDs.

The classical algorithm for building a threshold BDD (see e.g. [Weg00]) is in principle similar to dynamic programming for solving a Knapsack problem (see e.g. [Sch86]). It is a recursive method, which ensures a unique representation of the output by applying certain rules while building the BDD. In particular, isomorphic subgraphs will be detected *after* being built and then deleted or merged again. This raises our first question.

Question 1. *Can an algorithm be given, which only constructs as many nodes of the graph representation as the threshold BDD consists of?*

For many problems in combinatorial optimization there exists a *0/1 integer programming* (0/1 IP) formulation, i.e. a set of linear constraints together with a linear objective function and a restriction of the variables to 0 or 1. The natural way for building a BDD for such a problem is the following. First build a threshold BDD for each constraint separately, and then use a pairwise *and*-operator on the set of BDDs in a sequential fashion, until one BDD is left. This way, intermediate BDDs will be constructed which can have a representation size, that is several times larger than that of the final BDD. This severe problem motivates our next question.

Question 2. *Is there a different approach for the* and*-operation, such that the size explosion caused by intermediate BDDs can be avoided?*

Until now, we looked at the connection between BDDs and 0/1 integer programming only from one point of view. But we are also interested in the opposite direction.

Question 3. *How can 0/1 integer programming be applied to the field of threshold BDDs?*

In general, 0/1 integer programming problems are hard to solve although they might have a small representation size. The transformation of such a problem to a BDD shifts these properties, i.e. the representation size possibly gets large while the optimization problem becomes fairly easy to solve. In fact, it reduces to a shortest path problem on a directed acyclic graph which can be solved in linear time in the number of nodes of the graph. This is the main motivation for the investigation of using BDDs in 0/1 integer programming. Apart from optimization, a lot of other tasks can be efficiently tackled if the BDD for a 0/1 integer programming problem could be build.

In polyhedral studies of 0/1 polytopes two prominent problems exist. One is the *vertex enumeration problem*: Given a system of inequalities, count or enumerate its feasible 0/1 points. In addition, if a linear objective function is given, compute one optimal solution, or count or enumerate all optimal solutions.

Question 4. *How can these tasks be accomplished using BDDs?*

Another one is the *convex hull problem*: Given a set of 0/1 points in dimension d, enumerate the facets of the corresponding polytope.

Question 5. *How can BDDs be used for computing the convex hull of 0/1 polytopes?*

Branch & Cut is an effective method for solving 0/1 IPs. In theory it is also possible to solve such problems by building the according BDD. But the disadvantage of BDDs is, that building the entire BDD is in practice hard. The running time of Branch & Cut depends on many things, among which are the "quality" of separated cutting planes. A further point of interest is the generation of cutting planes from not only one constraint of the problem formulation but from two or an arbitrary set of constraints. This leads to the final question.

Question 6. *How can we combine advantages of both fields to develop a fast Branch & Cut algorithm using BDDs to generate cutting planes?*

1.2 Outline

This thesis focuses on the above questions, mainly from a practical point of view.

We first review the preliminaries in chapter 2. In particular, we assume that the reader is familiar with combinatorial optimization and integer programming but less familiar with binary decision diagrams.

In chapter 3 we develop a new output-sensitive algorithm for building a threshold BDD which answers question 1. More precisely, our algorithm constructs exactly as many nodes as the final BDD consists of and does not need any extra memory. Then we are concerned with question 2. We give an *and*-operation that synthesizes all threshold BDDs *in parallel* which is also a novelty. Thereby we overcome the problem of explosion in size during computation. Regarding question 3, we develop for the first time a 0/1 IP, whose optimal solution gives the size and the optimal variable order of a threshold BDD. Usually, the variable ordering spectrum of a BDD is not computable. With the help of this 0/1 IP, we are now able to compute the variable ordering spectrum of a threshold BDD.

In chapter 4 we are concerned with polyhedral aspects of 0/1 polytopes. We developed a tool called `azove`[1], which is capable of vertex counting, enumeration and optimization and thus solves the tasks raised by question 4 with the help of BDDs. Computational results show that our tool is currently the fastest available. On some instances, it is several orders of magnitude faster than existing codes. We also investigate the convex hull problem of 0/1 polytopes as raised by question 5. We extend the gift-wrapping algorithm with BDDs to solve the facet enumeration problem. As shown by computational results, our approach can be recommended for 0/1 polytopes whose facets contain few vertices.

Chapter 5 gives a detailed answer to question 6. We apply for the first time BDDs for separation in a Branch & Cut framework and develop all necessary methods. The computational results which we achieved on MAX-ONES instances and randomly generated 0/1 IPs show, that we developed code which is competitive with state-of-the-art MIP solvers.

1.3 Sources

The material and results in chapters 3 and 4 are from the papers [BE07] and [Beh07a, Beh07b]. The concepts and results in chapter 5 are from the paper [BBEW05].

[1]`http://www.mpi-inf.mpg.de/~behle/azove.html`, Another Zero One Vertex Enumeration homepage, M. Behle, 2007

2 Preliminaries

In the following we introduce terms and definitions in the way we will use them throughout this work. The sectioning is in dependence on the three main chapters:

3 Binary Decision Diagrams

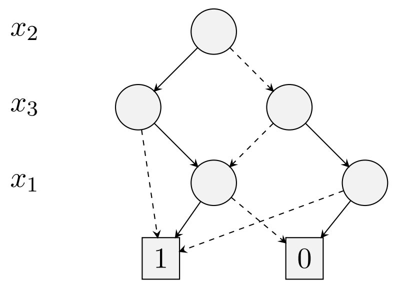
4 Polyhedral problems

5 Integer Programming

Since the integration of Binary Decision Diagrams into the different fields of polyhedral investigation, combinatorial optimization and integer programming is the main aspect of this work, the fundamental terms defined in section 2.1 are essential to know in each chapter.

2.1 Binary Decision Diagrams

This work heavily relies on *Binary Decision Diagrams* (BDDs for short), a datastructure which represents a set of 0/1 points in a compact way. We provide a definition of BDDs as they are used throughout this work. For further discussions on BDDs we refer the reader to the books [MT98, Weg00].



(a) BDD

x_1	x_2	x_3
0	0	1
0	1	0
1	0	0
1	1	0
1	1	1

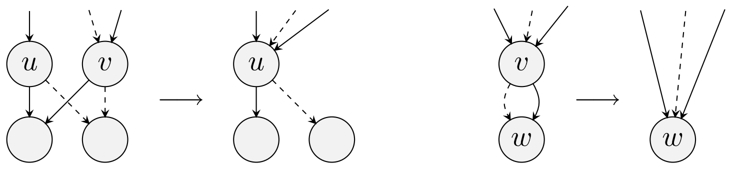
(b) Represented set T of true-assignments

Figure 2.1: A simple BDD represented as a directed graph. Edges with parity 0 are dashed. The variable order is x_2, x_3, x_1. The table shows the represented 0/1 points of the set T.

A *BDD* for a set of d variables $x_1, \ldots, x_d$ is a directed acyclic graph $G = (V, A)$. An example is given in figure 2.1(a). The graph has one node with in-degree zero, called the root and two nodes with out-degree zero, called leaf 0 resp. leaf 1. There is a labeling function $\ell\colon V \setminus \{\text{leaf } 0, \text{leaf } 1\} \to \{x_1, \dots, x_d\}$. All nodes labeled with the variable x_i lie on the same level, which means we have an *ordered* BDD (*OBDD*). Thus a level is associated with a variable x_i. This relation is given by the *variable order*, which is a permutation $\pi\colon \{1, \dots, d\} \to \{x_1, \dots, x_d\}$. In this work all BDDs are ordered. For convenience we only write BDD instead of OBDD.

For the edges there is a parity function $\text{par}\colon A \to \{0, 1\}$. Apart from the leaves all nodes have exactly two outgoing edges with different parity, called the 0-edge resp. the 1-edge according to their parity. Only edges with a direction from top to bottom (concerning the levels) are allowed. A path $e_1, \dots, e_d$ from the root to one of the leaves represents a variable assignment in such a way, that the label x_i of the head of e_j is assigned to the value $\text{par}(e_j)$. An edge crossing a level with nodes labeled x_i is called a *long* edge. In that case the assignment for x_i is free. If each path from the root to leaf 1 contains exactly d edges the BDD is called *complete*.

All paths from the root to leaf 1 represent the set $T \subseteq \{0, 1\}^d$ of true-assignments, whereas the paths from the root to leaf 0 represent the set $F \subseteq \{0, 1\}^d$ of false-assignments. We always have $T \mathbin{\dot\cup} F = \{0, 1\}^d$. Thus a BDD represents a partition of all 0/1 vertices of the unit hypercube in dimension d.



(a) Merging rule (b) Elimination rule

Figure 2.2: The two reduction rules for OBDDs.

Two vertices $u, v \in V$ are *equivalent*, if they have the same label and both of their edges with the same parity point to the same node respectively. A complete and ordered BDD with no equivalent vertices is called a *quasi-reduced* ordered BDD (*QOBDD*). A vertex $v \in V$ is *redundant*, if both outgoing edges point to the same node w. If an ordered BDD does neither contain equivalent nor redundant vertices it is called *reduced* ordered BDD (*ROBDD*). For a fixed variable order both QOBDD and ROBDD are canonical representations. In order to achieve such a canonical representation, the following two *reduction rules* are sufficient.

Merging rule: If two vertices u, v are equivalent, then eliminate v and redirect all incoming edges of v to u.

Elimination rule: If vertex v is redundant, then eliminate v and redirect all incoming edges of v to its successor w.

Figure 2.2 illustrates the two reduction rules for OBDDs. Obviously the elimination rule can be applied to a BDD in $\mathcal{O}(|V|)$. Bryant's approach [Bry86] to apply the merging rule needs $\mathcal{O}(|V| \log|V|)$ time. Sieling and Wegener [SW93] proposed a two phase bucket sort approach for the merging rule which has linear runtime $\mathcal{O}(|V|)$. However it can only be used after a BDD was built completely.

The *size* of a BDD is defined as the number of nodes $|V|$. Let w_l be the number of nodes in level l. The *width* of a BDD is the maximum of all numbers of nodes in a level $w = \max\{w_l \mid l \in 1, \dots, d\}$. Naturally $|V| \leq dw$ holds. A variable order is called *optimal* if it belongs to those variable orders for which the size of the BDD is minimal. The *variable ordering spectrum* of a linear constraint $a^{\mathrm{T}}x \leq b$ is the function $sp_{a^{\mathrm{T}}x \leq b} \colon \mathbb{N} \to \mathbb{N}_0$, where $sp_{a^{\mathrm{T}}x \leq b}(k)$ is the number of variable orderings leading to a BDD of size k for the threshold function $a^{\mathrm{T}}x \leq b$.

2.2 Polyhedral problems

Next we review some terminology from *polyhedral theory* that we need in our context in the corresponding chapter 4. Parts of the definitions given in this section are also used in chapter 5 when it comes to polyhedral aspects within integer programming. For a more detailed view on polytopes we refer the reader to the excellent book [Zie95].

First we recall the notion of some standard norms for vectors, i.e. the l_1- or *1-norm* $\|v\|_1 := \sum_{i=1}^{d} |v_i|$, the l_2- or *Euclidean norm* $\|v\| := \left(\sum_{i=1}^{d} v_i^2\right)^{\frac{1}{2}}$, and the l_∞- or *maximum norm* $\|v\|_\infty := \max_{1 \leq i \leq d} |v_i|$.

A vector $x \in \mathbb{R}^d$ is called a *linear combination* of the vectors $x_1, \dots, x_k \in \mathbb{R}^d$, if $x = \sum_{i=1}^{k} \lambda_i x_i$ for some $\lambda \in \mathbb{R}^k$. Additionally, if $\sum_{i=1}^{k} \lambda_i = 1$ holds, x is called an *affine combination*. Furthermore if $\sum_{i=1}^{k} \lambda_i = 1$ and $\lambda \geq 0$, x is a *convex combination* of the vectors $x_1, \dots, x_k$. Given a nonempty set $S \subseteq \mathbb{R}^d$, the set of all vectors which are affine resp. convex combinations of finitely many vectors of S are denoted as the *affine hull* $\operatorname{aff}(S)$ resp. *convex hull* $\operatorname{conv}(S)$ of S.

A *polyhedron* P is a set of vectors of the form $P = \{x \in \mathbb{R}^d \mid Ax \leq b\}$ for some matrix $A \in \mathbb{R}^{m \times d}$ and some vector $b \in \mathbb{R}^m$. The polyhedron is *rational* if both A and b can be chosen to be rational. If P is bounded, then P is called a *polytope*. The *integer hull* $P_{\mathrm{I}} := \operatorname{conv}(P \cap \mathbb{Z}^d)$ of a polytope P is the convex hull of the integral vectors in P.

The representation theorem (also called the main theorem) for polytopes states that a polytope P can be described by two independent characterizations, namely as the convex hull $P = \operatorname{conv}(S)$ of a finite point set S (a $\mathcal{V}$-polytope) and as the bounded intersection of halfspaces $P = \{x \in \mathbb{R}^d \mid Ax \leq b\}$ (an $\mathcal{H}$-polytope). Both characterizations are in a certain sense equivalent. This is "geometrically clear" but nontrivial to prove and we refer the interested reader for the details again to the book [Zie95].

The *dimension* $\dim(P)$ of P is the dimension of its affine hull $\dim(\operatorname{aff}(P))$. P is *full-dimensional* if $\dim(P) = d$. If P is not full-dimensional then at least one of the describing inequalities of $Ax \leq b$ is satisfied at equality by all points of P, and thus the *interior* of P is empty, i.e. $\operatorname{int}(P) = \emptyset$. Therefore we define the *relative interior* of P as the interior of P with respect to its embedding into its affine hull $\operatorname{aff}(P)$, in which P is full-dimensional. Note that the maximum number of affinely independent points in P is $\dim(P) + 1$ and that the vectors $v_0, v_1, \dots, v_i \in \mathbb{R}^d$ are affinely independent iff the vectors $v_1 - v_0, \dots, v_i - v_0 \in \mathbb{R}^d$ are linearly independent.

We define the origin of the vector space $\mathbb{R}^d$ as $\mathbf{0} \in \mathbb{R}^d$. If P is full-dimensional and $\mathbf{0} \in \text{int}(P)$ holds, the descriptions of P as a $\mathcal{V}$-polytope and an $\mathcal{H}$-polytope are equivalent under point/hyperplane duality. For this we need for any set $S \subseteq \mathbb{R}^d$ the notion of its *polar* S^* which is $S^* := \{y \in \mathbb{R}^d \mid y^\mathrm{T} x \leq 1 \text{ for all } x \in S\}$. The extension to that is the so-called *γ-polar* of S, which is $S^*_\gamma := \{(y^\mathrm{T}, \gamma)^\mathrm{T} \in \mathbb{R}^{d+1} \mid y^\mathrm{T} x \leq \gamma \text{ for all } x \in S\}$.

A *0/1 polytope* is a polytope, which is the convex hull of a set of 0/1 points $S \subseteq \{0,1\}^d$. Thus for a 0/1 polytope the notion of its 0/1 points and extremal points resp. vertices is the same since every vertex is a 0/1 point and vice versa. Naturally every 0/1 polytope is contained in the *unit hypercube* which is defined as $\{x \in \mathbb{R}^d \mid 0 \leq x_i \leq 1 \; \forall i \in \{1, \dots, d\}\}$.

An inequality $c^\mathrm{T} x \leq \delta$ with $c \in \mathbb{R}^d$ and $\delta \in \mathbb{R}$ is *valid* for P if it is satisfied by all points in P. The *faces* F of a convex polytope P are $\emptyset$, P and the intersection of a supporting hyperplane of P with P itself, i.e. $F = P \cap \{x \in \mathbb{R}^d \mid c^\mathrm{T} x = \delta\}$ with $c^\mathrm{T} x \leq \delta$ valid for P. Thus if $c^\mathrm{T} x \leq \delta$ is valid and $\delta = \max\{c^\mathrm{T} x \mid x \in P\}$, it defines the face $F = \{x \in P \mid c^\mathrm{T} x = \delta\}$ of P. The *dimension* of a face F of P is the dimension of its affine hull $\dim(\text{aff}(F))$. The face F is a *facet* of P, if $\dim(F) = \dim(P) - 1$. Faces of dimension 0, 1, $\dim(P) - 2$, and $\dim(P) - 1$ are called *vertices* (or *extreme points*), *edges*, *ridges*, and *facets* respectively.

A polytope P is called *simplicial*, if every facet contains exactly d vertices. P is called *simple*, if every vertex is the intersection of exactly d facets. The input for the facet enumeration problem is called *degenerate* if there are more than d points which lie on a common hyperplane, and *nondegenerate* otherwise.

The faces of a polytope P can be partially ordered by inclusion. Imagine a graph with node layers from $\dim(P)$ down to 0, where the nodes in layer i represent all i-dimensional faces of P. Thus P is on the top layer and $\emptyset$ on the bottom layer. An edge between a node in layer i and $i+1$ states that for the corresponding faces $F_i \supset F_{i+1}$ holds. The in this way constructed graph is a representation of the *face lattice* of the polytope P.

The *facet graph* of a polytope P is a graph, whose nodes represent the facets of the polytope P, and with two nodes adjacent by an edge, iff the corresponding facets share a common ridge.

Complexity

Dealing with the computational complexity of algorithms for enumeration problems, we consider both the size of the input and the output, since in general the output size might be exponential in the input size.

An algorithm is called *output-sensitive* if its runtime is bounded in terms of the output size as well as the input size. Implicit in describing an algorithm as output-sensitive is that the dependence on the output size is "reasonable", which usually means bounded by a small polynomial (see [Bre96]). Furthermore an enumeration algorithm is called *polynomial* if its runtime is polynomial in the size of the input and the output for all inputs. A *linear* algorithm is a polynomial algorithm, whose runtime is linear in the size of the output.

If the space complexity of an algorithm is polynomial in the input size and not depending on the output size, it is called *compact*. An enumeration problem is *strongly $\mathcal{P}$-enumerable*, if there exists a linear and compact algorithm which solves it.

2.3 Integer Programming

We will now give a short introduction to the concepts and terms that we need from the field of Linear and Integer Programming. This is by far not meant to be a comprehensive survey. More details, further aspects and a deeper insight into this subject can be gained from the book [Sch86].

A *Linear Programming (LP)* problem asks for an optimal solution of a linear objective function over a polyhedron P. It is usually given in one of its standard forms, i.e.

$$\begin{aligned} \max \quad & c^{\mathrm{T}} x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^d \end{aligned} \tag{2.1}$$

where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$ define the polyhedron P and the objective function is given by $c \in \mathbb{R}^d$. There are equivalent forms with $Ax = b$, $Ax \geq b$ or bounds $l_i \leq x_i \leq u_i$ on the variables x_i for $i \in \{1, \ldots, d\}$, but we will mainly use the formulation (2.1). In addition, the kind of optimization direction can equally be chosen between maximization or minimization. A fundamental result is that Linear Programming is in $\mathcal{P}$ (for details see e.g. [GLSv88]).

In *Integer Programming (IP)*, or more precise Integer Linear Programming, the task is to find an integer vector $x \in \mathbb{Z}^d$, which is an optimal solution to the problem defined by the linear objective function and linear constraints given in (2.1). Throughout this work, we will only examine the case of *0/1 Integer Programming (0/1 IP)*, i.e. we are interested in optimal solutions given by a binary vector $x \in \{0, 1\}^d$. Many combinatorial optimization problems are modeled with decision variables and thus can be formulated as 0/1 integer programs. We will use the following standard form for 0/1 IPs

$$\begin{aligned} \max \quad & c^{\mathrm{T}} x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0, 1\}^d \end{aligned} \tag{2.2}$$

where the matrix A and the vectors b and c are rational. 0/1 Integer Programming is $\mathcal{NP}$-complete. In case A and b define the integer hull P_{I} of a 0/1 polytope P, 0/1 Integer Programming reduces to Linear Programming. An inequality description of P_{I} however can be exponential.

A matrix A is called *totally unimodular*, if each subdeterminant of A is 0, 1 or -1, so in particular each entry in A is either 0, 1 or -1. There exist a lot of equivalent characterizations of total unimodularity for a matrix A. If the entries of A, b and c in (2.2) are integral and A is totally unimodular, then the *linear relaxation* $0 \leq x \leq 1$ of (2.2) has an integral optimum solution. So in case A is totally unimodular, the 0/1 IP (2.2) can be solved with Linear Programming. In our context we will use the known fact, that the node-edge incidence matrix of a directed graph is totally unimodular.

One of the basic algorithms for solving 0/1 IPs is *Branch & Bound*. It is an implicit enumeration of the solution space. The problem is decomposed into smaller subproblems by setting up two branches, with x_i fixed to 0 on one and fixed to 1 on the other branch. This recursively leads to a search tree, called the Branch & Bound tree. In addition, for each node a bound on the best solution of its branch is calculated, often via solving the LP relaxation under the given fixations. Depending on this bound or the detection of infeasibility, the node will be pruned or the algorithm further branches on it. Among other considerations, maintaining the list of active subproblems and the order of examination of these subproblems are important issues regarding the runtime. Note that a complete enumeration of the solution space is totally impossible for most problems, since there are 2^d possibilities.

Another concept to solve a 0/1 IP is the so-called *cutting plane algorithm*. Here the idea is to solve resp. reoptimize the associated linear programming relaxation. If the solution is fractional, new cutting planes which are valid for the 0/1 IP are found via calling a separation routine. The addition of these cutting planes to the LP relaxation then results in a better approximation of the integer hull. A general method is the application of Chvátal-Gomory cuts which can be generated from an optimal but fractional solution of the LP relaxation. The nature of the applied cutting planes, e.g. whether they are facet-defining for the 0/1 IP, is very import for the running time of this approach. Often the family of valid inequalities generated by a separation routine is enormous. The addition of all these cuts results in big linear programs, which take a long time to solve. Therefore the detection of "strong" cutting planes is important.

Branch & Cut incorporates ideas from both solution strategies. It is a Branch & Bound algorithm in which cutting planes are generated at the nodes to improve the linear relaxations. By this, the bound on the best solution of a branch can be tightened, and thus the number of nodes of the Branch & Bound tree can be reduced. In addition, not only cuts are used to improve the performance, but a lot of other techniques like primal heuristics, preprocessing at each node, use of cut pools, and so forth.

Integer programming solvers have become one of the most important industrial strength tools for solving applied optimization problems in the last years. Branch & Cut still is the most successful method for solving 0/1 integer programming problems. It is applied by all competitive commercial codes.

3 Binary Decision Diagrams

Binary decision diagrams (or *BDDs* for short) were first proposed by Lee in 1959 [Lee59] and further studied by Akers in 1978 [Ake78]. In 1986 Bryant [Bry86][1] extended BDDs significantly. He introduced a canonical representation using fixed variable ordering, used shared sub-graphs for a compacter representation and presented efficient algorithms for the synthesis of BDDs. After that, BDDs became very popular. Nowadays they are used as an effective datastructure in the area of hardware verification (e.g. VLSI design) and computational logics (e.g. model checking), see e.g. [MT98, Weg00].

In chapters 4 and 5 we extend the usage of BDDs to problems from the fields of polyhedral investigation, combinatorial optimization and integer programming. In particular we use the fact that 0/1 integer programs are related to knapsack, subset sum and multidimensional knapsack problems. Building a BDD for these problems incorporates building a BDD for a linear constraint, a so-called *weighted threshold BDD*. In section 3.1.2 we present a novel approach for this task which consists of an *output-sensitive* algorithm for building a threshold BDD. Combining linear constraints relates to an *and*-operation on threshold BDDs. We provide a *parallel and*-operation on threshold BDDs in section 3.2.2.

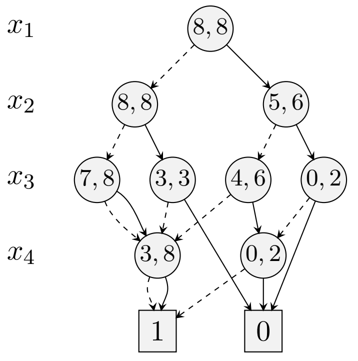
BDDs are represented as a directed acyclic graph. The size of a BDD is the number of nodes of its graph. It heavily depends on the chosen variable order. Bollig and Wegener showed in [BW96] that improving a given variable order is $\mathcal{NP}$-complete. So finding the optimal variable order is an $\mathcal{NP}$-hard problem. In section 3.3.5 we derive a 0/1 integer program for finding an optimal variable order of a threshold BDD. With the help of this formulation the computation of the variable order spectrum of a threshold function is possible (see section 3.3.6).

3.1 Weighted threshold BDDs

Definition 3.1. *A BDD representing the set $T = \left\{x \in \{0,1\}^d \mid a^{\mathrm{T}} x \leq b\right\}$ of 0/1 solutions to the linear constraint $a^{\mathrm{T}} x \leq b$ is called a* weighted threshold BDD.

The width of a threshold BDD only depends on the weights. For each variable order it is obviously bounded by $\mathcal{O}\left(|a_1| + \ldots + |a_d|\right)$. Therefore the size of a threshold BDD is bounded by $\mathcal{O}\left(d(|a_1| + \ldots + |a_d|)\right)$. If the weights $a_1, \ldots, a_d$ are polynomial bounded in

[1] As of September 2006 this paper is the most cited paper according to CiteSeer.IST, Scientific Literature Digital Library.



(a) BDD

x_1	x_2	x_3	x_4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
1	0	0	0
1	0	0	1
1	0	1	0
1	1	0	0

(b) Represented set T of true-assignments

Figure 3.1: A threshold BDD representing the characteristic function of the linear constraint $2x_1 + 5x_2 + 4x_3 + 3x_4 \leq 8$. Edges with parity 0 are dashed.

d, the size of the BDD is polynomial bounded in d. For any variable order the size of a threshold BDD is known to be upper bounded by $\mathcal{O}\left(2^{d/2}\right)$ [HTKY97].

For a long time it was not clear whether all threshold BDDs can be represented in polynomial size with an adaptively chosen variable order. Hosaka et al. [HTKY97] provided an example of an explicitly defined threshold function for which the size of the BDD is exponential for all variable orders. Be k even and $d = k^2$. The linear constraint is defined on the d variables x_{ij}, $1 \leq i, j \leq k$. Be $a_{ij} = 2^{i-1} + 2^{k+j-1}$. Then for any variable order the size of the threshold BDD for the linear constraint $\sum_{1 \leq i,j \leq k} a_{ij} x_{ij} \geq k(2^{2k} - 1)/2$ is lower bounded by $\Omega\left(d2^{\sqrt{d}/2}\right)$. An alternative proof to the one given in [HTKY97] can be found in [Weg00].

Definition 3.1 reveals the close relation between building a threshold BDD and solving the *knapsack problem*

$$\max\{c^{\mathrm{T}}x \mid a^{\mathrm{T}}x \leq b,\ a \in \mathbb{Z}^d,\ b \in \mathbb{Z},\ x \in \{0,1\}^d\}$$

resp. the *subset sum problem*

$$\exists x \in \{0,1\}^d :\ a^{\mathrm{T}}x = b,\ a \in \mathbb{Z}^d,\ b \in \mathbb{Z}.$$

In the following sections we will compare our techniques for building threshold BDDs with the dynamic programming approaches for the knapsack problem presented in [Sch86].

3.1.1 Basic construction

Consider the function $f\colon \{0,1\}^d \to \mathbb{Z}$ with $f(x) := a^{\mathrm{T}}x - b$, $a \in \mathbb{Z}^d$, $b \in \mathbb{Z}$. Algorithm 3.1 shows how to build the BDD for the characteristic function of the linear constraint $f(x) \leq 0$. It is similar to a dynamic programming approach. We start at the root of

Algorithm 3.1 Build BDD for $f(x) \leq 0$

```
    BuildBDD($f$)
 1: if $\max(f) \leq 0$ then
 2:     return leaf 1
 3: if $\min(f) > 0$ then
 4:     return leaf 0
 5: if $f \in$ ComputedTable then
 6:     return ComputedTable[f]
 7: $x_i$ = NextVariable(f)
 8: BDD low = BuildBDD($f|_{x_i=0}$)
 9: BDD high = BuildBDD($f|_{x_i=1}$)
10: BDD result = $x_i \cdot$ high + $\bar{x}_i \cdot$ low
11: ComputedTable[f] = result
12: return result
```

the BDD and traverse it in a depth-first-search manner. For every node we recursively construct its both sons and then build the node itself.

Let a given variable order be fixed. Define $a^+ := \sum_{a_i>0} a_i$ and $a^- := \sum_{a_i<0} a_i$. We set up a table of size $d \times (a^+ - a^-)$ in which we save results (step 11) and look up already computed parts of the BDD in constant time (step 5). To start building the BDD we call BuildBDD($a^\mathrm{T}x - b$). First we check for trivial cases (steps 1 and 3). Note that it is sufficient to compute the global minimum and maximum for f once. These are $a^- - b$ resp. $a^+ - b$. All other local minima and maxima can be computed in constant time by the recursive calls. After the selection of a variable x_i according to the given variable order (step 7) we build the children of the actual node with restriction of the variable x_i to 0 resp. 1 (steps 8 - 9). In step 10 a new node will be inserted on top of both children. This is done by the so-called *If-Then-Else-operator (ITE)*. It is a ternary Boolean function with inputs x, h, l that computes the function *If x, then h, else l*

$$ITE(x, h, l) = x \cdot h + \bar{x} \cdot l$$

The ITE-operator reflects *Shannon's decomposition rule* performed in a node of the BDD:

$$f(x) = x_i \cdot f|_{x_i=1} + \bar{x}_i \cdot f|_{x_i=0}$$

The number of variables fixed to a value determines the level l on which an operation is performed. In a node on level l be $\bar{a} := \sum_{x_i=1} a_i$ the sum of all a_i for which x_i has been fixed to 1. In step 5 the table will then be accessed at position $l \times (\bar{a} - a^-)$. Note that this look-up is not exact, i.e. there exist nodes on a level l which have different values $\bar{a}$ but are equivalent.

Algorithm 3.1 can be adapted to build the BDD for a linear equation $f(x) = 0$. This relates to the subset sum problem. The following slight modifications have to be applied:

```
1: replace $\max(f) \leq 0$ with $\min(f) = 0 \wedge \max(f) = 0$
```

```
3: replace $\min(f) > 0$ with $\min(f) > 0 \lor \max(f) < 0$
```

Let $\|a\|_\infty$ be the maximum absolute value of all components of the vector a. Then the following lemma holds.

Lemma 3.1. *The runtime and the space complexity for building a BDD for the characteristic function of a linear constraint $a^\mathrm{T}x \le b$ are both $\mathcal{O}\left(d^2\|a\|_\infty\right)$.*

Proof. We set up a table of size $d \times (a^+ - a^-)$. The size of the BDD cannot exceed the table size. With $d\|a\|_\infty \ge a^+ - a^-$ the space complexity holds.

Without the recursive calls in steps 8 and 9 the algorithm has constant runtime. So the runtime only depends on the number of recursive calls. Because of the table the algorithm is never called twice for the same look-up $l \times (\bar{a} - a^-)$. Thus the runtime is bounded by the space complexity. □

Hence the runtime and space complexity for building a BDD for the characteristic function of a linear constraint are both pseudo-polynomial. In section 4.2 we will see that optimizing according to a linear function $c^\mathrm{T}x$ can be done in linear time in the size of the BDD. So the knapsack problem can be solved using a BDD in time $\mathcal{O}\left(d^2\|a\|_\infty\right)$.

The same holds for the dynamic programming approach to the knapsack problem. Here a directed graph of size $(d+1) \times (2d\|a\|_\infty + 1)$ is also levelwise allocated (cf. [Sch86]). From each level l to the next level $l+1$ there are two kinds of edges: edges of type $(l, \delta) \to (l+1, \delta)$ with costs 0 and edges of type $(l, \delta) \to (l+1, \delta + a_l)$ with costs c_l. Any directed path from $(0,0)$ to (d, b') with $b' \le b$ yields a feasible solution. An optimal solution can be computed by finding a shortest path from $(0,0)$ to (d, b') for some $b' \le b$ in linear time in the size of the graph.

The advantage of the BDD approach over the dynamic programming approach however is the use of the ComputedTable. Its leads to a compacter representation of all feasible solutions as isomorphic subgraphs can be detected and reused.

3.1.2 Output-sensitive building

A crucial point of BDD construction algorithms is the in advance detection of equivalent nodes (cf. [MT98]). If equivalent nodes are not fully detected this leads to isomorphic subgraphs. As the representation of QOBDDs and ROBDDs is canonical these isomorphic subgraphs have to be detected and merged at a later stage which is a considerable overhead. In this section we describe a new output-sensitive algorithm that overcomes this drawback. Given a linear constraint $a^\mathrm{T}x \le b$ in dimension d it builds the threshold QOBDD of its characteristic functions. Our detection of equivalent nodes is exact and complete so that only as many nodes will be built as the final QOBDD consists of. No nodes have to be merged later on.

W.l.o.g. we assume $\forall i \in \{1, \dots, d\}\ a_i \ge 0$ (in case $a_i < 0$ substitute x_i with $1 - \bar{x}_i$). In order to exclude trivial cases let $b \ge 0$ and $\sum_{i=1}^d a_i > b$. For the sake of simplicity be the given variable order the canonical variable order $x_1, \dots, x_d$. We assign weights to the edges depending on their parity and level. Edges with parity 1 in level l cost a_l and edges with parity 0 cost 0. The key to exact detection of equivalent nodes are two bounds that we introduce for each node, a lower bound lb and an upper bound ub. They describe the interval $[lb, ub]$. Let c_u be the costs of the path from the root to the node u. All nodes u in level l for which the value $b - c_u$ lies in the interval $[lb_v, ub_v]$ of a node v in level l are guaranteed to be equivalent with the node v. We call the value $b - c_u$ the slack. Figure 3.1(a) illustrates a threshold QOBDD with the intervals set in each node.

Algorithm 3.2 Build QOBDD for the constraint $a^{\mathrm{T}}x \leq b$

```
    BuildQOBDD(slack, level)
 1: if slack $< 0$ then
 2:     return leaf 0
 3: if slack $\geq \sum_{i=\text{level}}^{d} a_i$ then
 4:     return leaf 1
 5: if exists node $v$ in level with $lb_v \leq$ slack $\leq ub_v$ then
 6:     return $v$
 7: build new node $u$ in level
 8: $l$ = level of node
 9: 0-edge son = BuildQOBDD(slack, $l+1$)
10: 1-edge son = BuildQOBDD(slack - $a_l$, $l+1$)
11: set lb to max(lb of 0-edge son, lb of 1-edge son + $a_l$)
12: set ub to min(ub of 0-edge son, ub of 1-edge son + $a_l$)
13: return $u$
```

Algorithm 3.2 constructs the QOBDD top-down from a given node in a depth-first-search manner. We set the bounds for the leaves as follows: $lb_{\text{leaf }0} = -\infty$, $ub_{\text{leaf }0} = -1$, $lb_{\text{leaf }1} = 0$ and $ub_{\text{leaf }1} = \infty$. We start at the root with its slack set to b. While traversing downwards along an edge in step 9 and 10 we subtract its costs. The sons of a node are built recursively. The slack always reflects the value of the right hand side b minus the costs c of the path from the root to the node. In step 5 a node is detected to be equivalent with an already built node v in that level if there exists a node v with slack $\in [lb_v, ub_v]$.

If both sons of a node have been built recursively at step 11 the lower bound is set to the costs of the longest path from the node to leaf 1. In case one of the sons is a long edge pointing from this level l to leaf 1 the value $lb_{\text{leaf }1}$ has to be temporarily increased by $\sum_{i=l+1}^{d} a_i$ before. In step 12 the upper bound is set to the costs of the shortest path from the node to leaf 0 minus 1. For this reason the interval $[lb, ub]$ reflects the widest possible interval for equivalent nodes.

Lemma 3.2. *The in advance detection of equivalent nodes in algorithm 3.2 is exact and complete.*

Proof. Assume to the contrary that in step 7 a new node u is built which is equivalent to an existing node v in the level. Again let c_u be the costs of the path from the root to the node u. Because of step 5 we have $b - c_u \notin [lb_v, ub_v]$.
Case $b - c_u < lb_v$:
In step 11 lb_v has been computed as the costs of the longest path from the node v to leaf 1. Let lb_u be the costs of the longest path from node u to leaf 1. Then there is a path from

root to leaf 1 using node u with costs $c_u + lb_u \leq b$, so we have $lb_u < lb_v$. As the nodes u and v are equivalent they are the root of isomorphic subtrees, and thus $lb_u = lb_v$ holds.
Case $b - c_u > ub_v$:
With step 12 ub_v is the costs of the shortest path from v to leaf 0 minus 1. Be ub_u the costs of the shortest path from u to leaf 0 minus 1. Again the nodes u and v are equivalent so for both the costs we have $ub_u = ub_v$. Thus there is a path from root to leaf 0 using node u with costs $c_u + ub_u < b$ which is a contradiction. □

Algorithm 3.2 can be modified to work for a given equation, i.e. it can also be used to solve the subset sum problem. The following replacements have to be made:

```
1: replace slack $< 0$ with slack $< 0 \vee$ slack $> \sum_{i=\text{level}}^{d} a_i$
3: replace slack $\geq \sum_{i=\text{level}}^{d} a_i$ with slack $= 0 \wedge$ slack $= \sum_{i=\text{level}}^{d} a_i$
```

Be n the size the threshold QOBDD. We can then formulate the following corollary.

Corollary 3.1. *The space complexity of algorithm 3.2 is $\mathcal{O}(n \log(b))$.*

Proof. Because of lemma 3.2 exactly n nodes are created. The merging rule is not needed. In every node we save two non-negative values which are bounded by b. □

In contrast to the basic construction algorithm and the dynamic programming approach discussed in section 3.1.1 the space complexity does not depend on the values of the input but on the input and output size.

Let w be the width of the QOBDD which is naturally bounded by n. The following lemma states that our algorithm is output-sensitive.

Lemma 3.3. *The runtime of algorithm 3.2 is $\mathcal{O}(n \log(w))$.*

Proof. Searching for an equivalent node in step 5 can be done in time $\log(w)$. Without step 5 and the recursive calls in steps 9 and 10 the algorithm has constant runtime. Thus the runtime only depends on the number of recursive calls. Because of lemma 3.2 there are $2n$ recursive calls. □

The size n of the QOBDD is bounded by dw and in addition the width w is bounded by $d\|a\|_\infty$. So an upper bound of the runtime is $\mathcal{O}\left(d^2\|a\|_\infty \log(w)\right)$. In order to construct a QOBDD with the basic construction algorithm and the dynamic programming approach presented in section 3.1.1, the merging rule needs to be applied afterwards. For each node an equivalent node can be found in time $\log(w)$ with the help of dynamic search trees. Then with lemma 3.1 the resulting runtime is the same as the upper bound $\mathcal{O}\left(d^2\|a\|_\infty \log(w)\right)$. Sieling and Wegener [SW93] showed how to apply the merging rule to a non-reduced BDD in time linear in the size of the non-reduced BDD. Their approach is based on a two phase bucket sort. However it cannot be used for the in advance detection of equivalent nodes.

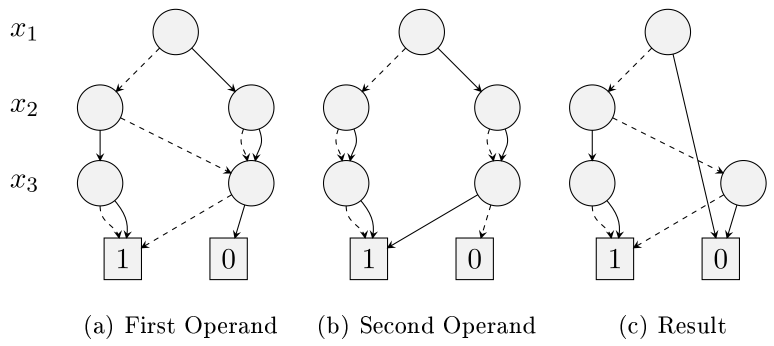


Figure 3.2: Conjunction of BDDs: The first operand is the BDD of the characteristic function of the constraint $2x_1 - x_2 + 3x_3 \leq 2$, the second of $x_1 - x_3 \leq 0$.

3.2 Synthesis

Now that we know how to build the BDD for a single linear constraint or equation we will tackle *0/1 integer programming* problems given in the following form

$$\max\{c^{\mathrm{T}}x \mid Ax \leqq b,\ A \in \mathbb{Z}^{m\times d},\ b \in \mathbb{Z}^m,\ x \in \{0,1\}^d\}. \tag{3.1}$$

Some or all of the m constraints are allowed to be equations. Although we do not restrict A to be non-negative we consider the input $Ax \leqq b$ as a *multidimensional knapsack problem.* Our approach consists of two steps. For every of the m constraints construct the BDD of its characteristic function. After that build the conjunction of the BDDs. The final BDD represents the set of 0/1 solutions to the system $Ax \leqq b$. Figure 3.2 shows an example of a conjunction of two BDDs. With the technique described in section 4.2 the optimization problem can then be solved in time linear in the size of the final BDD.

We describe two ways for the conjunction of BDDs. The classical approach is a pairwise conjunction on the set of BDDs until one final BDD is left. The size of the intermediate BDDs can be significantly larger than the size of the final BDD. Our new method of conjunction avoids this explosion in size by performing an *and*-operation on all threshold BDDs in parallel.

3.2.1 Sequential *and*-operation

Let f and g be two boolean functions, e.g. characteristic functions of linear constraints. Be $G_f = (V_f, A_f)$ resp. $G_g = (V_g, A_g)$ their graph representations as BDDs. Algorithm 3.3 describes a straight forward recursive approach for the synthesis of two BDDs with the binary operator *and* (see e.g. [Weg00]).

We start at the root of both BDDs G_f and G_g. The top-most variable is set to 0 resp. 1 and for both BDDs the algorithm is called recursively on the two branches. Via the use of the ComputedTable in steps 5 and 11 we push the detection and reusage of isomorphic subgraphs.

Algorithm 3.3 Conjunction of the two BDDs G_f and G_g

```
    andBDDs($G_f$, $G_g$)
 1: if $G_f$ = leaf 1 $\wedge$ $G_g$ = leaf 1 then
 2:     return leaf 1
 3: if $G_f$ = leaf 0 $\vee$ $G_g$ = leaf 0 then
 4:     return leaf 0
 5: if $(G_f, G_g) \in$ ComputedTable then
 6:     return ComputedTable[$(G_f, G_g)$]
 7: $x_i$ = NextVariable($(G_f, G_g)$)
 8: BDD low = andBDDs($G_f|_{x_i=0}$, $G_g|_{x_i=0}$)
 9: BDD high = andBDDs($G_f|_{x_i=1}$, $G_g|_{x_i=1}$)
10: BDD result = $x_i\cdot$ high + $\bar{x}_i\cdot$ low
11: ComputedTable[$(G_f, G_g)$] = result
12: return result
```

Lemma 3.4. *The binary synthesis of the two BDDs G_f and G_g with the operator* and *is possible in time and space $\mathcal{O}(|V_f||V_g|)$.*

Proof. Since all steps can be performed in constant runtime only the number of recursive calls is important. The number of reachable nodes in $G_f \times G_g$ determines the maximal size of the computed table and thus the runtime. □

In practice the typical performance is closer to the size of the resulting BDD which is often smaller than $|V_f||V_g|$. Note that algorithm 3.3 can be used for the synthesis of any kind of BDDs, not only threshold BDDs.

Now we return to the 0/1 integer programming problem (3.1). Assume that for all of the m constraints a threshold BDD has been built. Again we point out that these m constraints can be equations or inequalities. Let set B consist of these m BDDs. We iteratively pick two BDDs from B, compute their conjunction with algorithm 3.3 and put the result back in B until one BDD is left in B. This final BDD then represents the characteristic function of the system $Ax \leqq b$. Let $A_{\max}$ be the maximum absolute value of all components of the matrix A. Then lemma 3.1 and the sequential use of the lemma 3.4 leads to the following corollary.

Corollary 3.2. *The runtime and the space complexity for the conjunction of m threshold BDDs defined by the system of linear inequalities $Ax \leqq b$ are both $\mathcal{O}\left((d^2 A_{\max})^m\right)$.*

Given a BDD and a linear objective function, optimization can be performed in time linear in the size of the BDD, see section 4.2. So for any fixed number of constraints m the 0/1 integer programming problem (3.1) can be solved in pseudo-polynomial time.

It is known that the integer programming problem with $Ax = b$, $x \geq 0$ and $x \in \mathbb{Z}^d$ can be solved in pseudo-polynomial time for any fixed number of constraints [Pap81]. If $Ax = b$ has a solution $x \geq 0$ it also has one with entries bounded by dM with $M = (mA_{\max})^{2m+1}$. The dynamic programming approach given in [Pap81] has runtime $\mathcal{O}\left((d^2 M)^{m+1}\right)$.

The order of the BDDs chosen from B for the pairwise conjunction decides on the size of the intermediate BDDs. In order to keep the size of each conjunction small it is advisable to choose the two smallest BDDs. Even though the size of the final BDD is small an explosion in the sizes of the intermediate BDDs may not be prevented in general.

3.2.2 Parallel *and*-operation

Our goal is to circumvent the explosion in size while building the final BDD. Therefore we abstain from using intermediate BDDs by constructing the final BDD right from the beginning. Given a set of inequalities $Ax \leqq b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$, we want to build the ROBDD representing all 0/1 points satisfying the system. For each of the m linear constraints let the threshold QOBDDs be built with the method described in section 3.1.2. Then we build the final ROBDD by performing an *and*-operation on all threshold QOBDDs in parallel. The space consumption for saving the nodes is exactly the number of nodes that the final ROBDD consists of plus d temporary nodes. Algorithm 3.4 describes our parallel *and*-synthesis of m QOBDDs.

Algorithm 3.4 Parallel conjunction of the QOBDDs $G_1, \dots, G_m$

```
    parallelAndBDDs($G_1, \dots, G_m$)
 1: if $\forall i \in \{1, \dots, m\} : G_i =$ leaf 1 then
 2:     return leaf 1
 3: if $\exists i \in \{1, \dots, m\} : G_i =$ leaf 0 then
 4:     return leaf 0
 5: if signature($G_1, \dots, G_m$) $\in$ ComputedTable then
 6:     return ComputedTable[signature($G_1, \dots, G_m$)]
 7: $x_i =$ NextVariable($G_1, \dots, G_m$)
 8: 0-edge son = parallelAndBDDs($G_1|_{x_i=0}, \dots, G_m|_{x_i=0}$)
 9: 1-edge son = parallelAndBDDs($G_1|_{x_i=1}, \dots, G_m|_{x_i=1}$)
10: if 0-edge son = 1-edge son then
11:     return 0-edge son
12: if $\exists$ node $v$ in this level with same sons then
13:     return $v$
14: build node $u$ with 0-edge and 1-edge son
15: ComputedTable[signature($G_1, \dots, G_m$)] = $u$
16: return $u$
```

We start at the root of all QOBDDs and construct the ROBDD from its root top-down in a depth-first-search manner. In steps 1 and 3 we check in parallel for trivial cases. Next we generate a signature for this temporary node of the ROBDD in step 5. This signature is a $1 + m$ dimensional vector consisting of the current level and the upper bounds saved in all current nodes of the QOBDDs. If there already exists a node in the ROBDD with the same signature we have found an equivalent node and return it. Otherwise we start building both sons recursively from this temporary node in steps 8 and 9. From all starting nodes in the QOBDDs we traverse the edges with the same parity in parallel.

When both sons of a temporary node in the ROBDD were built we check its redundancy in step 10. In step 12 we search for an already existing node in the current level which is equivalent to the temporary node. If neither is the case we build this node in the ROBDD and save its signature.

Be n the size of the final ROBDD. The following lemma states that algorithm 3.4 prevents an explosion in the size needed for the construction.

Lemma 3.5. *Algorithm 3.4 needs $n + d$ nodes for the construction of the ROBDD plus additional space for the ComputedTable.*

Proof. For every of the d levels a temporary node is needed. In a level a node of the ROBDD will only be built if it is not equivalent to an existing node. The size of the ComputedTable for saving the signatures is bounded by the number of reachable nodes in $G_1 \times \ldots \times G_m$. $\square$

Now be w the width of the final ROBDD. Assume that enough space is available for storing the complete ComputedTable with size $\prod_{i=1}^{m}|G_i|$. Then we have the following lemma.

Lemma 3.6. *The runtime of algorithm 3.4 is $\mathcal{O}\left((m + \log(w)) \prod_{i=1}^{m}|G_i|\right)$.*

Proof. For the checks in steps 1 and 3 and the computation of the signature in step 5 all QOBDDs have to be accessed. Hence the runtimes of these operations are $\mathcal{O}(m)$. The look-up and the insert in the ComputedTable in steps 5 and 15 and the check for redundancy in step 10 are possible in constant time. Searching an equivalent node in step 12 can be accomplished in $\mathcal{O}(\log(w))$. The main factor for the runtime is the number of recursive calls. These are bounded by the maximum possible size of the ComputedTable which is $\prod_{i=1}^{m}|G_i|$. $\square$

In practice the main problem of the parallel *and*-operation is the low hitrate of the ComputedTable. This is because equivalent nodes of the ROBDD can have different signatures and thus are not detected in step 5. In addition the space consumption for the ComputedTable is enormous and one is usually interested in restricting it. The space available for saving the signatures in the ComputedTable can be changed dynamically. This controls the runtime in the following way. The more space is granted for the ComputedTable the more likely equivalent node will be detected in advance which decreases the runtime. Note that because of the check for equivalence in step 12 the correctness of the algorithm does not depend on the use of the ComputedTable. If the use of the ComputedTable is little the algorithm naturally tends to exponential runtime.

Nevertheless the advantage of algorithm 3.4 in comparison to algorithm 3.3 is that the size of the final ROBDD is an exact limit on the space needed for the construction.

3.3 Variable order

It is well-known that the variable order used in a BDD has a great influence on its size (cf. [Weg00]). For BDDs representing a threshold function the assignment of variables with larger weights has a high impact on the weighted sum of the input. Therefore it is likely that the descending order of the absolute values of the weights yields a variable order for which the size of the threshold BDD is small. Hosaka et al. [HTKY97] provided an example which is contrary to this intuition. Given a positive even number d and the linear constraint $\sum_{i=1}^{d/2} 2^{i-1}x_i + \sum_{i=d/2+1}^{d}(2^{d/2} - 2^{d-i})x_i \geq 2^{d/2}d/4$. The size of the corresponding threshold BDD is bounded below by $\binom{d/2}{d/4}$ if the variables are ordered according to descending weights $x_d x_{d-1} \dots x_1$, whereas for the variable order $x_1 x_d x_2 x_{d-1} \dots x_{d/2} x_{d/2+1}$ the upper bound for the size is $\mathcal{O}\left(d^2\right)$.

Nevertheless in practice the total order of weights is a good indicator for choosing a variable order which might lead to a small size of the BDD. Finding a variable order for which the size of a BDD is minimal is a difficult task. Bollig and Wegener [BW96] showed that improving a given variable order of a general BDD is $\mathcal{NP}$-complete. Thus it is a $\mathcal{NP}$-hard problem to find an optimal variable order. We derive a 0/1 integer program in section 3.3.5 whose optimal solution gives the minimal size and an optimal variable order for the threshold BDD of a given linear constraint. In section 3.3.6 this formulation is the basis for the computation of the variable order spectrum of a threshold function.

3.3.1 Pre-construction heuristics

Before we start to build the BDD for a set of constraints $Ax \leqq b$ we have to choose an initial variable order. This variable order should preferably lead to a BDD with small size. Experiments have shown that heuristics which do not take the structure of the problem into account tend to produce bad variable orders. We developed two heuristics which consider all constraints $Ax \leqq b$ and compute an appropriate initial variable order. Both consist of three steps. First the constraint set $Ax \leqq b$ is partitioned into subsets, the so-called *blocks*. Then for every block a partial variable order is computed. In the last step these partial variable orders are merged into one global variable order. The two heuristics only differ in the way they divide up the set of constraints into blocks.

For the partitioning of the constraints in the first step we adapt an algorithm which is designed for partitioning the outputs of circuits [HKB04]. Define the *support* of a constraint over the variables $x_1, \dots, x_d$ as the set of variable indices with nonzero coefficients, i.e. $\operatorname{supp}(a^{\mathrm{T}}x \leq b) := \{i \in \{1, \dots, d\} \mid a_i \neq 0\}$. We sort the constraints in decreasing order of the size of their support. Now start with a new initially empty block. The constraint with the largest support is deleted from the set of constraints and inserted into the new block. This constraint is called the *leader* of the block. Then all constraints satisfying a certain criterion are moved to the new block. If there are still constraints remaining we construct a new empty block and iterate the procedure until the set of constraints becomes empty. We use two different criteria to determine if a constraint belongs to a block, the *Word-oriented Output Grouping* and the *Bit-oriented Output Grouping*:

- WOG: Add a constraint if its support is a subset of the support of the leader.
- BOG: Add a constraint if its support is a subset of the supports of all constraints already contained in the block.

The idea behind both criteria is to group constraints with similar support as they can share the same structure in the BDD. In [HKB04] it is shown that the WOG heuristic tends to generate less blocks than the BOG heuristic.

Once the blocks were built we use a simple heuristic to compute the partial orders for every block. Be $A'x \leqq b'$ the set of constraints belonging to a block. For every variable x_j in the block we compute the sum of the absolute values of its coefficients $w_j := \sum_{i=1}^{m'} |a'_{ij}|$ and then sort the variables in decreasing order of their w_j value. This partial variable order reflects that variables with larger coefficients likely have a higher impact on the structure of the BDD.

Before we construct the global variable order with the help of the partial variable orders we sort the blocks increasingly by the number of variables contained in them. Then we merge the partial variable orders given by the blocks using a technique called *interleaving* [FOH93]. Given the global variable order and a block with a partial variable order that we want to merge into it. The interleaving algorithm works in the following way. Check every variable x_i in the block in the order given by the partial variable order if it is already contained in the global variable order. If this is the case proceed to the next variable. Otherwise determine its predecessor x_j in the partial variable order and insert x_i in the global variable order behind x_j. If x_i is the topmost variable of the partial variable order insert it at the top of the global variable order. Thus the interleaving algorithm preserves the structures of the partial variable orders within the global variable order.

3.3.2 Sifting algorithm

In section 3.2.1 we have observed that during the sequential conjunction of BDDs the size of the intermediate BDDs might grow drastically. It is desirable to reduce the size after a certain limit in size is exceeded. Among the methods to improve the variable order by dynamic reordering (cf. [MT98] for an overview) is the well-known *sifting algorithm* by Rudell [Rud93]. It can be applied after the construction of a general BDD. We will describe the algorithm in the following.

The sifting algorithm is based on the *swap*-operator which locally exchanges the order of two successive variables. W.l.o.g. be the variable order canonical $x_1, \ldots, x_d$. Then the swap operation on the variable x_i exchanges x_i with its successor x_{i+1} in the BDD. This local reordering only affects the levels i and $i+1$ and the runtime is linear in the number of nodes of both levels. An efficient implementation of the swap-operator is described in [MT98].

Next we use the swap operation to find a locally optimum position for a variable x_i assuming that all other variables remain fixed. Start at the current level of x_i and move x_i down by swapping it with its successor until x_i reaches the last level d. Then use the swap-operator to move x_i upwards until it is at the root. During the down and upward movements of x_i we find the position j with the smallest size of the BDD. At last we use the swap-operator to move x_i from the root to that level j. In order to reduce the number of swap operations the first moving direction can be chosen adaptively, i.e. if x_i is closer to the root it will first be moved upwards and afterwards down to the last level.

The sifting algorithm now works as follows. First sort the variables in decreasing order based on the number of nodes in the current level of the variable. According to this sorting then find for each variable a locally optimum position. We start with the variable which occurs most as a label of a node since it possesses the largest optimization potential.

Be w the width of the BDD. The algorithm needs $\mathcal{O}\left(d^2\right)$ swap-operations which have a complexity of $\mathcal{O}\left(w\right)$ each. As the runtime of the sorting is dominated by the number of swap-operations the runtime of the sifting algorithm is $\mathcal{O}\left(d^2 w\right)$.

3.3.3 Size reduction with unused constraints

Our aim is to build the BDD for a set of constraints $Ax \leq b$. Now assume that we have only achieved to build the BDD for a subset $A'x \leq b'$ of the constraints $Ax \leq b$ so far. Possible reasons are that we did not finish the sequential conjunction (see section 3.2.1) yet or that we could not finish it because the memory or time consumption is too high. So there are some constraints left in the set $Ax \leq b$ which are not included in $A'x \leq b'$ and thus have not been used for building the current BDD. In the following we show how to apply these *unused* constraints to decrease the size of the BDD.

Let $a^{\mathrm{T}}x \leq b$ be such an unused constraint. For each edge in the BDD we set the following weights (compare (4.3) in section 4.2):

$$w(e) = \begin{cases} a_i & \text{if } \operatorname{par}(e) = 1 \text{ and } \ell(\operatorname{head}(e)) = x_i \\ 0 & \text{if } \operatorname{par}(e) = 0 \text{ otherwise} \end{cases}$$

For all nodes $v \in V$ of the BDD we compute the length $l_{\downarrow}(v)$ of the shortest path starting from the root to it. Next we compute the length $l_{\uparrow}(v)$ of the shortest path from leaf 1 upwards to all nodes v. Then the value $l(v) := l_{\downarrow}(v) + l_{\uparrow}(v)$ determines the length of the shortest path from the root to leaf 1 crossing node v. We observe the following:

- If there exists a node v with $l(v) > b$ all paths crossing this node represent vectors $\hat{x} \in \{0,1\}^d$ with $a^{\mathrm{T}}\hat{x} > b$. $\Longrightarrow$ Delete v and redirect all incoming edges to leaf 0.

For all edges $e \in A$ we can determine the length of the shortest path from the root to leaf 1 using edge e as $l'(e) := l_{\downarrow}(\operatorname{head}(e)) + w(e) + l_{\uparrow}(\operatorname{tail}(e))$. Then the extension of our observation reads as follows:

- If there exists an edge e with $l'(e) > b$ all paths using this edge represent vectors $\hat{x} \in \{0,1\}^d$ with $a^{\mathrm{T}}\hat{x} > b$. $\Longrightarrow$ Redirect the tail of e to leaf 0.

As the graph representation of the BDD is acyclic the shortest path computations run in linear time in the size of the BDD.

These reductions can be applied to the BDD for all unused constraints in $Ax \leq b$. Depending on the structure of $a^{\mathrm{T}}x \leq b$ and $A'x \leq b'$ the size of the BDD reduces considerably.

3.3.4 Exact minimization

In the following we give a survey of the development of techniques for the exact minimization of BDDs. All algorithms aim at computing a variable order for which the size of the

BDD is minimal. They can be applied to all types of BDDs. Nearly all of them are based on the classic method by Friedman and Supowit [FS90] and continuously improve on each other.

Friedman and Supowit [FS90] gave the first algorithm to find an optimal variable order. Instead of trying all $d!$ permutations in a naive way in time $\mathcal{O}\left(d!\, 2^d\right)$ they developed a dynamic programming approach with a significantly better runtime of $\mathcal{O}\left(d^2\, 3^d\right)$. Their algorithm heavily depends on their fundamental lemma.

Lemma 3.7. *Let $I \subseteq \{x_1, \ldots, x_d\}$ be a subset of all variables with cardinality $k = |I|$ and be $x_i \in I$. Then there exists a constant c such that the number of nodes labeled with x_i equals c for all variable orders given by a permutation $\pi\colon \{1, \ldots, d\} \to \{x_1, \ldots, x_d\}$ with $\{\pi(1), \ldots, \pi(k)\} = I$ and $\pi(k) = x_i$.*

Informally this means that the number of nodes in a level is constant, if the corresponding variable is fixed in the variable order and no variables from the lower and upper part are exchanged. This holds independently of the variable orders in the upper and lower part. With the help of this key fact the entries of the tables in the dynamic programming algorithm can be computed as follows. Be $I \subseteq \{x_1, \ldots, x_d\}$ fixed with $k = |I|$. Assume that we know for all $I' \subset I$ with $|I'| = k - 1$ the variable orders for the first $k - 1$ variables which lead to a minimum number of nodes with labels from I'. Add the variable $x_i \in I \setminus I'$ in level k to all of them. Then the minimum number of nodes with labels from I and the according variable order for the first k elements can be found. So the optimal variable order can be computed iteratively by computing for increasing k the number of nodes and variable orders for all k-element subsets I until $k = d$.

Ishiura et al. [ISY91] improved Friedman and Supowit's approach. The explicit construction of tables for storing the results of subproblems is omitted. Instead all relevant subfunctions are represented by BDDs. They additionally were the first to use Branch & Bound. For pruning the search space they use the following simple lower bound. Assume that the BDD is built from bottom to top. For the last k variables the labels be from the k-element set $I \subseteq \{x_1, \ldots, x_d\}$. Let the minimum number of nodes and an according variable order for I be already known. On level $d-k+1$ there be c nodes labeled with $x_i \in I$. This implies that there have to be at least $c-1$ nodes in the part of the BDD above level $d-k+1$. Thus increasing the number of labels in set I to $k+1$ the minimal number of nodes for the new I is at least the minimum number of nodes for I plus $c-1$.

A better lower bound was presented by Jeong et al. [JKS93]. Moreover the exchange of variables for the construction of the BDDs for the sets I was performed more efficiently which led to an increased performance of the algorithm.

The next step in the chain of improvements was developed by Drechsler et al. [DDG98]. They used a new lower bound from the field of circuit complexity theory and very large scale integration (VLSI) design, which was presented by Bryant [Bry91]. This is the tightest lower bound known today.

Up to that point all approaches used one lower bound. Ebendt et al. [EGD03] generalized the lower bound of [Bry91] in different ways and then extended their approach with a combination of three lower bounds in parallel.

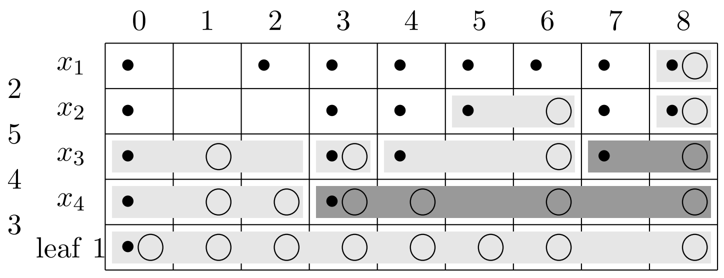


Figure 3.3: Dynamic programming table for the linear constraint $2x_1+5x_2+4x_3+3x_4 \leq 8$. Variables U_{ln}, D_{ln} are shown as $\bullet$, $\bigcirc$ resp. The light grey blocks represent the nodes in the ROBDD, and the dark grey blocks represent the redundant nodes in the QOBDD.

The last improvement in the series of exact minimization algorithms based on Friedman and Supowit's method was again achieved by Ebendt [Ebe03]. The expensive movements of variables through the BDD were substituted by a state expansion technique which significantly reduced the runtime.

In [EGD04] Ebendt et al. broke new ground. They did not use Friedman and Supowit's approach any longer but combined the search algorithm A^* known from artificial intelligence with Branch & Bound. They reused the lower bound they had developed in [EGD03] and the state expansion technique presented in [Ebe03].

All of the known algorithms have in common that they need to build a BDD for the computation of an optimal variable order.

3.3.5 0/1 Integer Programming

Given a linear constraint $a^{\mathrm{T}}x \leq b$ in dimension d we want to find an optimal variable order for building the corresponding threshold ROBDD. In the following we derive a 0/1 integer program whose solution gives the optimal variable order and the minimal number of nodes needed. It also forms the basis for the computation of the variable order spectrum of a threshold function in section 3.3.6. In contrast to all other exact BDD minimization techniques (see section 3.3.4), our approach does not need to build a BDD explicitly.

As we have seen in section 3.1, building a threshold BDD is closely related to solving a knapsack problem. A knapsack problem can be solved with dynamic programming [Sch86] using a table. We mimic this approach on a virtual table of size $(d+1) \times (b+1)$ which we fill with variables. Figure 3.3 shows an example of such a table for a fixed variable order. The corresponding BDD is shown in figure 3.1(a) on page 12.

W.l.o.g. we assume $\forall i \in \{1, \dots, d\}$ $a_i \geq 0$, and to exclude trivial cases, $b \geq 0$ and $\sum_{i=1}^{d} a_i > b$. Now we start setting up the 0/1 IP shown in figure 3.4. The 0/1 variables y_{li} (3.25) encode a variable order in the way that $y_{li} = 1$ iff the variable x_i lies on level l. To ensure a correct encoding of a variable order we need that each index is on exactly one level (3.3) and that on each level there is exactly one index (3.4).

We simulate a *down operation* in the dynamic programming table with the 0/1 variables D_{ln} (3.26). The variable D_{ln} is 1 iff there exists a path from the root to the level l such

$$
\begin{array}{rcr}
\min & \sum_{\substack{l\in\{1,\dots,d+1\}\\ n\in\{0,\dots,b\}}} C_{ln} + 1 & (3.2)\\
\text{s.t.} & & \\
\forall i \in \{1,\dots,d\} & \sum_{l=1}^{d} y_{li} = 1 & (3.3)\\
\forall l \in \{1,\dots,d\} & \sum_{i=1}^{d} y_{li} = 1 & (3.4)\\
\forall n \in \{0,\dots,b-1\} & D_{1n} = 0 & (3.5)\\
\forall l \in \{1,\dots,d+1\} & D_{lb} = 1 & (3.6)\\
\forall n \in \{1,\dots,b\} & U_{(d+1)n} = 0 & (3.7)\\
\forall l \in \{1,\dots,d+1\} & U_{l0} = 1 & (3.8)\\
 & B_{(d+1)0} = 1 & (3.9)\\
\forall n \in \{1,\dots,b\} & B_{(d+1)n} = 0 & (3.10)\\
 & C_{(d+1)0} = 1 & (3.11)\\
\forall n \in \{1,\dots,b\} & C_{(d+1)n} = 0 & (3.12)\\
 & & \\
\forall l \in \{1,\dots,d\}\colon & & \\
\forall n \in \{0,\dots,b-1\} & D_{ln} - D_{(l+1)n} \le 0 & (3.13)\\
\forall n \in \{1,\dots,b\} & U_{(l+1)n} - U_{ln} \le 0 & (3.14)\\
\forall n \in \{0,\dots,b\}, j \in \{1,\dots,n+1\} & D_{ln} + U_{l(j-1)} - \sum_{i=j}^{n} U_{li} - B_{l(j-1)} \le 1 & (3.15)\\
 & & \\
\forall l \in \{1,\dots,d\}, i \in \{1,\dots,d\}\colon & & \\
\forall n \in \{0,\dots,b-a_i\} & y_{li} + D_{l(n+a_i)} - D_{(l+1)n} \le 1 & (3.16)\\
\forall n \in \{b-a_i+1,\dots,b-1\} & y_{li} - D_{ln} + D_{(l+1)n} \le 1 & (3.17)\\
\forall n \in \{0,\dots,b-a_i\} & y_{li} - D_{l(n+a_i)} - D_{ln} + D_{(l+1)n} \le 1 & (3.18)\\
\forall n \in \{a_i,\dots,b\} & y_{li} + U_{(l+1)(n-a_i)} - U_{ln} \le 1 & (3.19)\\
\forall n \in \{1,\dots,a_i-1\} & y_{li} - U_{(l+1)n} + U_{ln} \le 1 & (3.20)\\
\forall n \in \{a_i,\dots,b\} & y_{li} - U_{(l+1)(n-a_i)} - U_{(l+1)n} + U_{ln} \le 1 & (3.21)\\
\forall n \in \{0,\dots,a_i-1\} & y_{li} + B_{ln} - C_{ln} \le 1 & (3.22)\\
\forall n \in \{0,\dots,a_i-1\} & y_{li} - B_{ln} + C_{ln} \le 1 & (3.23)\\
\forall n \in \{a_i,\dots,b\}, k \in \{n-a_i+1,\dots,n\} & y_{li} + B_{ln} + B_{(l+1)k} - C_{ln} \le 2 & (3.24)\\
 & & \\
\forall l \in \{1,\dots,d\}, i \in \{1,\dots,d\}\colon & y_{li} \in \{0,1\} & (3.25)\\
\forall l \in \{1,\dots,d+1\}, n \in \{0,\dots,b\}\colon & D_{ln}, U_{ln} \in \{0,1\} & (3.26)\\
 & B_{ln}, C_{ln} \in \{0,1\} & (3.27)
\end{array}
$$

Figure 3.4: 0/1 integer program for finding the optimal variable order of a threshold BDD for a linear constraint $a^{\mathrm{T}}x \le b$ in dimension d.

that b minus the costs of the path equals n. The variables in the first row (3.5) and the right column (3.6) are fixed. We have to set variable $D_{(l+1)n}$ to 1 if we followed the 0-edge starting from $D_{ln} = 1$

$$D_{ln} = 1 \rightarrow D_{(l+1)n} = 1 \quad (3.13)$$

or according to the variable order given by the y_{li} variables, if we followed the 1-edge starting from $D_{l(n+a_i)} = 1$

$$y_{li} = 1 \wedge D_{l(n+a_i)} = 1 \rightarrow D_{(l+1)n} = 1 \quad (3.16)$$

In all other cases we have to prevent $D_{(l+1)n}$ from being set to 1

$$\begin{aligned} y_{li} = 1 \wedge D_{ln} = 0 \rightarrow D_{(l+1)n} = 0 \quad (3.17) \\ y_{li} = 1 \wedge D_{l(n+a_i)} = 0 \wedge D_{ln} = 0 \rightarrow D_{(l+1)n} = 0 \quad (3.18) \end{aligned}$$

In the same way, the *up operation* is represented by the 0/1 variables U_{ln} (3.26). The variable U_{ln} is 1 iff there exists a path upwards from the leaf 1 to the level l with costs n. The variables in the last row (3.7) and the left column (3.8) are fixed. We have to set $U_{ln} = 1$ if there is a 0-edge ending in $U_{(l+1)n} = 1$

$$U_{(l+1)n} = 1 \rightarrow U_{ln} = 1 \quad (3.14)$$

or according to the variable order given by the y_{li} variables, if there is a 1-edge ending in $U_{(l+1)(n-a_i)} = 1$

$$y_{li} = 1 \wedge U_{(l+1)(n-a_i)} = 1 \rightarrow U_{ln} = 1 \quad (3.19)$$

In all other cases we have to prevent U_{ln} from being set to 1

$$\begin{aligned} y_{li} = 1 \wedge U_{(l+1)n} = 0 \rightarrow U_{ln} = 0 \quad (3.20) \\ y_{li} = 1 \wedge U_{(l+1)(n-a_i)} = 0 \wedge U_{(l+1)n} = 0 \rightarrow U_{ln} = 0 \quad (3.21) \end{aligned}$$

Next we introduce the 0/1 variables B_{ln} (3.27) which mark the beginning of the blocks in the dynamic programming table that correspond to the nodes in the QOBDD. These blocks can be identified as follows: start from a variable D_{ln} set to 1 and look to the left until a variable U_{ln} set to 1 is found

$$D_{ln} = 1 \wedge U_{l(j-1)} = 1 \wedge \bigwedge_{i=j}^{n} U_{li} = 0 \rightarrow B_{l(j-1)} = 1 \quad (3.15)$$

We set the last row explicitly (3.9), (3.10).

At last we introduce the 0/1 variables C_{ln} (3.27) which indicate the beginning of the blocks that correspond to the nodes in the ROBDD. The variables C_{ln} only depend on the B_{ln} variables and exclude redundant nodes, i.e. they introduce long edges. The first blocks are never redundant

$$y_{li} = 1 \rightarrow B_{ln} = C_{ln} \quad (3.22), (3.23)$$

If the 0-edge leads to a different block than the 1-edge, the block is not redundant

$$y_{li} = 1 \land B_{ln} = 1 \land \left(\bigvee_{k=n-a_i+1}^{n} B_{(l+1)k} = 1 \right) \to C_{ln} = 1 \quad (3.24)$$

We set the last row explicitly (3.11), (3.12).

The objective function (3.2) is to minimize the number of variables C_{ln} set to 1 plus an offset of 1 for counting the leaf 0. An optimal solution to the IP then gives the minimal number of nodes needed for the ROBDD while the y_{li} variables encode the best variable order.

In practice solving this 0/1 IP is not faster than the known exact BDD minimization algorithms (for an overview, see section 3.3.4). Nevertheless it is of theoretical interest as the presented 0/1 IP formulation can be used for the computation of the variable ordering spectrum of a threshold function.

3.3.6 Variable order spectrum of a threshold function

The *variable ordering spectrum* of a boolean function $\mathcal{B}\colon \{0,1\}^d \to \{0,1\}$ is the function $sp_{\mathcal{B}}\colon \mathbb{N} \to \mathbb{N}$, where $sp_{\mathcal{B}}(k)$ denotes the number of variable orderings leading to a ROBDD for the boolean function $\mathcal{B}$ of size k. Usually one is unable to compute or estimate this spectrum (see [Weg00]).

In contrast to that we can compute the variable ordering spectrum of a threshold function $a^{\mathrm{T}}x \leq b$ with the help of the 0/1 IP formulation given in figure 3.4. In order to compute $sp_{a^{\mathrm{T}}x\leq b}(k)$ we equate the objective function (3.2) with k and add it as the constraint $\sum_{\substack{l\in\{1,\ldots,d+1\}\\ n\in\{0,\ldots,b\}}} C_{ln} + 1 = k$ to the 0/1 IP formulation. The number of vertices of the corresponding 0/1 polytope then equals $sp_{a^{\mathrm{T}}x\leq b}(k)$. In section 4.3 we provide a method for counting these 0/1 vertices.

4 Polyhedral problems

In combinatorial optimization an important part in understanding and designing algorithms for a certain problem is the investigation of the polyhedral structure of the associated polytope. For many problems in this field the underlying polytope is a 0/1 polytope, i.e. all vertices are 0/1 points. Let the problem be given by a set of inequalities $Ax \leq b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$. The corresponding polytope P is then denoted by $P = \left\{x \in \mathbb{R}^d \mid Ax \leq b,\ 0 \leq x \leq 1\right\}$.

Be $G = (V, A)$ the graph representation of a BDD, which we built for the system of linear constraints $Ax \leq b$, and be P_{BDD} the polytope associated with it. In section 4.1 we investigate the relation between the polytope P_{BDD} and the flow-polytope of the graph G. The description P is a relaxation of P_{BDD} in the sense, that $P_{\text{BDD}} \subseteq P$ holds. As we have built the BDD for all constraints $Ax \leq b$, the BDD-polytope is equal to the integral hull of the problem, i.e. we have $P_{\text{BDD}} = P_{\text{I}}$.

The *membership problem* for a given 0/1 point $\hat{x} \in \{0,1\}^d$ and the polytope P is to decide whether $\hat{x} \in P$ holds. From the way of constructing the BDD follows, that for every binary point $\hat{x}$ we have $\hat{x} \in P$ if and only if $\hat{x} \in P_{\text{BDD}}$. Thus, by following the path in the BDD from the root according to the parity of the edges given by $\hat{x}$ to a leaf, this problem can be solved in $\mathcal{O}(d)$.

One of the most important problems in our context is *optimization*:

> Given a linear objective function by a vector $c \in \mathbb{R}^d$, compute an extremal point of P_{BDD} which maximizes resp. minimizes it.

In section 4.2 we show how to solve this problem in time *linear* in the size of the BDD. This key fact is used over and over throughout this work. With the help of fast optimization we are able to compute the *affine hull* of P_{BDD} (see section 4.2.1), determine the *dimension of a face* for P_{BDD} (in section 4.2.2) and decide the *polytope inclusion* problem (see section 4.2.3). We are also able to compute *all* optimal points with the same approach. In addition, the algorithms for building a threshold BDD can be extended to become *certifying* algorithms (see 4.2.4).

Another prominent problem is the *solution counting* problem:

> Given a set of inequalities $Ax \leq b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$, count the number of all 0/1 points satisfying the system.

This can be done with the help of the BDD without explicit enumeration in time linear in the size of the BDD, see section 4.3. In section 4.3.1 we show how to extend the approach in order to compute the *center of the polytope* P_{BDD} in the same time.

In section 4.4 we tackle one frequently arising problem, the *0/1 vertex enumeration* problem:

> Given a set of inequalities $Ax \leq b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$, compute a list of all 0/1 points satisfying the system.

In other words, if P is a 0/1 polytope, one is interested in the vertices of the *integer hull* P_I of P which generate the convex hull of all integer points of P.

In this work we present our tools `azove 1.1` and `2.0` which solve the 0/1 vertex counting and enumeration problems. Both outperform the currently best codes for these tasks by several orders of magnitude. In section 4.5 we introduce our tools and present computational results on benchmarks from the literature which show the strength of our new methods.

Another problem which we consider is the *facet enumeration* problem:

> Given a set $S \subseteq \{0,1\}^d$ of 0/1 points, enumerate all facets of the convex hull $\mathrm{conv}(S)$.

We develop in section 4.6 a gift-wrapping approach to solve the facet enumeration problem. Here we build the BDD for the set S and use it to rotate a facet-defining inequality along a ridge to find a new facet. Ridges are computed with existing codes. Computational results (in section 4.6.3) show that our approach can be recommended for polytopes whose facets contain few vertices.

A successful approach to difficult optimization problems with integer programming often requires some understanding of the facets of the integer hull of the solution space. A software package which computes the inequality representation $A_I x \leq b_I$ of the integer hull P_I of P, given an inequality representation $Ax \leq b$ of P, can here become very useful. Such an inequality representation is currently computed in a two-step approach. In a first step, one solves the 0/1 vertex enumeration problem, and then in a second step the facet enumeration problem for the previously generated 0/1 points is solved. With the help of our tools we can now omit the explicit enumeration step. First we build the BDD for the inequality set $Ax \leq b$ and then we can immediately start computing the facet description of the integer hull.

4.1 Relation between BDD-polytope and flow-polytope

We show the relation between the BDD-polytope (P_{BDD}) and the flow-polytope of the BDD graph (P_{flow}) which we define in the following.

Let $\mathcal{P}$ be the set of paths in the BDD from the root to the leaf 1. Each $p \in \mathcal{P}$ corresponds to an assignment of truth values to the d variables x_i via a characteristic vector $\chi_p \in \{0,1\}^d$, where $(\chi_p)_i = \mathrm{par}(e)$ with $e \in p$ and x_i is the head of e, and vice versa.

Definition 4.1 (BDD-polytope)**.** *The polytope given by a BDD can be described as*

$$P_{\mathrm{BDD}} = \mathrm{conv}(\chi_p \mid p \in \mathcal{P})$$

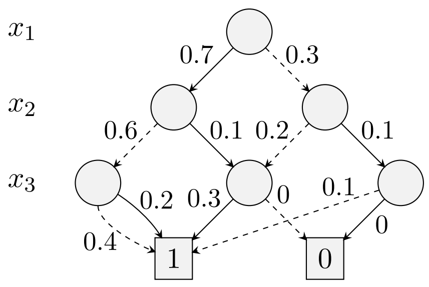


Figure 4.1: A flow corresponding to the assignment $x_1 = 0.7$, $x_2 = 0.2$ and $x_3 = 0.5$.

Now we enhance the graph $G = (V, A)$ of the BDD to a flow-network. Be $m = |A|$ the number of edges. We assign flow variables f_e to every edge e (for more details on network flows see [AMO93]). We use the root as a source with outflow 1. The sink is the leaf 1 with an inflow of 1. For all other nodes we want the outflow to be the same as the inflow. This leads to the following linear description.

Definition 4.2 (Flow-polytope)**.**

$$P_{\text{flow}} = \Big\{ \; f \in \mathbb{R}^m \mid \begin{array}{rl} \displaystyle\sum_{e\in\delta^+(v)} f_e - \sum_{e\in\delta^-(v)} f_e = 0 & \forall\, v \in V \setminus \{\text{root}, \text{leaf } 1\}, \\ \displaystyle\sum_{e\in\delta^+(\text{root})} f_e = 1 & \\ \displaystyle\sum_{e\in\delta^-(\text{leaf } 1)} f_e = 1 & \\ 0 \leq f_e \leq 1 & \forall\, e \in A \Big\} \end{array} \tag{4.1}$$

Note that P_{BDD} lives in d-dimensional space whereas P_{flow} lives in m-dimensional space.

Lemma 4.1. *The polytopes P_{BDD} and P_{flow} are both 0/1 polytopes with the same number of vertices respectively.*

Proof. As the node-edge incidence matrix of the flow-polytope is totally unimodular, each vertex of P_{flow} is binary and corresponds to a path from the source to the sink, i.e. a flow with $f_e \in \{0, 1\}$ $\forall e \in A$. For every such path there exists an incidence vector $f \in \{0, 1\}^m$ and a unique truth assignment for the variables $x_1, \dots, x_d$ via the characteristic vector $\chi_{\text{supp}(f)}$ where $\text{supp}(f) := \{e \in A \mid f_e \neq 0\}$.

On the other hand every truth assignment for $x_1, \dots, x_d$ corresponds to a unique path from the root to the leaf 1. Therefore there is a 1-1 correspondence between the vertices of the polytopes P_{BDD} and P_{flow}, which means that both have the same number of vertices. □

We need to provide linear equations which translate a feasible flow to its corresponding (fractional) assignment for the variables $x_1, \dots, x_d$. If f is the incidence vector of a path, then the i-th component of its corresponding truth assignment for x can be computed by

$$x_i = \sum_{e\colon \text{head}(e)=x_i} \text{par}(e) f_e. \tag{4.2}$$

We call (4.2) the *level equation* for the level with label x_i. Figure 4.1 illustrates how a flow corresponds to the assignment of the x-variables.

By adding the equations (4.2) for each $i = 1, \dots, d$ to the inequalities (4.1) we obtain a polytope $P(f,x) \subseteq \mathbb{R}^{m+d}$. The projection of $P(f,x)$ onto $\mathbb{R}^d$ is defined as

$$\text{Proj}_x(P(f,x)) := \{x \in \mathbb{R}^d \mid \exists f \in \mathbb{R}^m : (f,x) \in P(f,x)\},$$

which are all x for which there exists a flow fulfilling the level equations. Note that there might exist several flows which lead to the same assignment for x. Below we argue that the projection $\text{Proj}_x(P(f,x))$ of $P(f,x)$ onto the x-space is the polytope P_{BDD}.

Theorem 4.1.

$$P_{\text{BDD}} = \text{Proj}_x(P(f,x))$$

Proof. With lemma 4.1 we index the sets of vertices of the BDD-polytope and the flow-polytope with $j \in J$, so that the vertex $x^j \in P_{\text{BDD}}$ corresponds to the vertex $f^j \in P_{\text{flow}}$ and vice versa.

Let (f^*, x^*) be a feasible point of $P(f,x)$. The flow f^* can be written as a convex combination of the vertices f^j, $j \in J$ of the flow-polytope, i.e. $f^* = \sum_{j\in J} \mu_j f^j$ with $\forall j \in J : \mu_j \geq 0$ and $\sum_{j \in J} \mu_j = 1$. We have

$$\begin{aligned} x_i^* &= \sum_{e\colon \text{head}(e)=x_i} \text{par}(e) f_e^* \\ &= \sum_{e\colon \text{head}(e)=x_i} \text{par}(e) \big(\sum_{j\in J} \mu_j f^j\big)_e \\ &= \sum_{j\in J} \mu_j \sum_{e\colon \text{head}(e)=x_i} \text{par}(e) f_e^j \end{aligned}$$

For every path f^j there is only one edge with $\text{head}(e) = x_i$. So we have

$$x_i^* = \sum_{j\in J} \mu_j \, (\chi_{f^j})_i.$$

Hence it follows that $x^* \in P_{\text{BDD}}$.

Now be $x^* \in P_{\text{BDD}}$. It can be written as a convex combination of the vertices of the BDD-polytope

$$x^* = \sum_{j\in J} \mu_j x^j$$

with $\forall j \in J : \mu_j \geq 0$, $\sum_{j \in J} \mu_j = 1$. For every truth assignment x^j there exists a path f^j with $x^j = \chi_{f^j}$. We construct a flow $f := \sum_{j \in J} \mu_j f^j$. Now

$$\begin{aligned} x_i^* &= \sum_{j \in J} \mu_j x_i^j \\ &= \sum_{j \in J} \mu_j (\chi_{f^j})_i. \end{aligned}$$

□

In section 4.6.1 we use this theorem to compute the convex hull of P_{BDD}.

4.2 Optimization

In the following we deal with the import problem of optimization in connection with BDDs.

Definition 4.3 (BDD-OPT)**.** *Given a vector $c \in \mathbb{R}^d$ and a BDD by its graph representation $G = (V, A)$, compute a 0/1 point $\bar{x} \in P_{\text{BDD}}$ which is minimal w.r.t. the linear objective function $c^{\mathrm{T}} x$.*

As P_{BDD} is a 0/1 polytope the notion of 0/1 points and extremal points resp. vertices of P_{BDD} are the same. The following theorem states the key fact, that optimizing over the polytope P_{BDD} according to a linear objective function $c \in \mathbb{R}^d$ reduces to solving a shortest path problem on the directed acyclic graph G.

Theorem 4.2. *BDD-OPT can be solved in time linear in the size of the BDD.*

Proof. W.l.o.g. be the BDD complete. Set the edge weights on G to $w \colon E \to \mathbb{R}$, where

$$w(e) = \begin{cases} c_i & \text{if } \operatorname{par}(e) = 1 \text{ and } \ell(\operatorname{head}(e)) = x_i, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

With lemma 4.1 the optimal solutions to BDD-OPT are exactly the 0/1 points which are represented by a shortest path from root to leaf 1. Since G is acyclic, the shortest path problem can be solved in linear time $\mathcal{O}(|V|)$ as follows. For every node $v \in V$, initialize the costs of the shortest path from the root to it with ∞, except for the root, where we start from with costs 0. Now process all nodes v in the descending order of their level $1, \ldots, d$. If the costs of a successor node w of v are greater than the costs of v plus the edge costs $w(e)$ for $e := v \to w$, update the costs of w and set the predecessor of w to v. At the end, the costs of leaf 1 reflect the value of the shortest path. The corresponding 0/1 point $\bar{x}$ can be constructed by following the path of predecessors starting at leaf 1 up to the root. □

The problem of computing a 0/1 point which is *maximal* w.r.t. the linear objection function $c^{\mathrm{T}} x$ reduces in an analog way to finding a *longest* path in the directed acyclic graph G, which is also possible in time $\mathcal{O}(|V|)$. Solely the costs of the nodes have to be initialized with $-\infty$, and the costs of a successor node w will be updated if its costs are *less* than the costs of its predecessor v plus the corresponding edge costs $w(e)$.

Additionally it is also possible to compute *all* optimal $0/1$ points in time linear in the size of the BDD (without their explicit enumeration). Instead of saving one predecessor of a node w, we save all relevant predecessors. For this purpose the update step has to be modified as follows. If the costs of a successor node w of v are equal to the costs of v plus the corresponding edge costs $w(e)$, add v to the list of predecessors of w. Otherwise if the costs of w are greater resp. less than the costs of v plus $w(e)$, clear the list of predecessors of w and just insert v. Then all optimal $0/1$ points can be constructed by a recursive enumeration of the paths following all predecessors from leaf 1 upwards to the root.

4.2.1 Affine hull

Now that we know how to solve the optimization problem efficiently we can use it to determine the *affine hull*, the *dimension* and a *point in the relative interior* of P_{BDD}:

Definition 4.4. *Given the polytope P_{BDD} via a BDD, compute a system of affinely independent vertices $V := \{v_i \in P_{\text{BDD}} \mid i \in \{1,\dots,k\}\}$ such that*

$$\operatorname{aff}(P_{\text{BDD}}) = \operatorname{aff}(V)$$

and, in case P_{BDD} is not full-dimensional, a system of linearly independent equations $C := \{c_j^{\mathrm{T}} x = \gamma_j \mid \{1,\dots,d+1-k\}\}$ such that

$$\operatorname{aff}(P_{\text{BDD}}) = \{x \in \mathbb{R}^d \mid c_j^{\mathrm{T}} x = \gamma_j \ \forall j \in \{1,\dots,d+1-k\}\}.$$

Note that $|V| + |C| = d + 1$ holds. We have

$$\dim(P) = |V| - 1$$

and a point $\hat{x}$ in the relative interior of P_{BDD} is given by

$$\hat{x} := \frac{1}{|V|} \sum_{i=1}^{|V|} v_i$$

For the sake of completeness we describe an algorithm presented in [GLSv88] which was designed for the above purpose.

We choose a vertex $\bar{x}$ of P_{BDD} by traversing an arbitrary path from the root to leaf 1 in the BDD. Then we start with $V = \{\bar{x}\}$ and $C = \emptyset$. Assume now that we have already found affinely independent vertices V and linearly independent equations C. If $|V| + |C| = d + 1$ we stop. Otherwise we compute a basis of the subspace orthogonal to $\operatorname{aff}(V)$ and choose an equation c from it which is linearly independent from C. Then we maximize and minimize according to $c^{\mathrm{T}} x$ over P_{BDD} and find two optimal vertices x' and x'' of P_{BDD}. In case $c^{\mathrm{T}} x' = c^{\mathrm{T}} x''$ holds, we have found a new equation c with right hand side $c^{\mathrm{T}} x'$ and add it to C. Otherwise at least one of the vertices x' and x'' is not contained in $\operatorname{aff}(V)$. If $x' \in \operatorname{aff}(V)$ we add x'' to V, and otherwise x'.

Note that the tasks of finding a basis of the subspace orthogonal to $\operatorname{aff}(V)$ resp. choosing an equation c from this basis which is linearly independent from C and checking if $x' \in \operatorname{aff}(V)$ can be accomplished by solving a homogeneous system of linear equations resp. computing the rank of matrices.

4.2.2 Dimension of a face

Given a normal vector $c \in \mathbb{R}^d$, we want to determine the according right hand side γ such that $c^{\mathrm{T}}x \leq \gamma$ is valid for P_{BDD} and $F := P_{\mathrm{BDD}} \cap \{x \in \mathbb{R}^d \mid c^{\mathrm{T}}x = \gamma\}$ describes a face of P_{BDD}. In addition we want to know the *dimension of the face F*.

For this purpose we maximize according to $c^{\mathrm{T}}x$ over P_{BDD} and save all optimal points in the set of vertices V. The value of the longest path computation gives the according right hand side γ. Be $V = \{v_0, v_1, \ldots, v_k\}$. For $|V| \geq 2$, the dimension of the face F is then equal to the rank of the matrix given by $(v_1 - v_0, \ldots, v_k - v_0)$.

4.2.3 Polytope inclusion

The next decision problem, which we can solve with the help of efficient optimization, is the *polytope inclusion* problem:

Definition 4.5. *Given a polytope $P' := \{x \in \mathbb{R}^d \mid A'x \leq b',\ A' \in \mathbb{R}^{m' \times d},\ b' \in \mathbb{R}^{m'}\}$ and the polytope P_{BDD} via a BDD, decide whether $P_{\mathrm{BDD}} \subseteq P'$ holds.*

The approach works as follows. For every row $a'_{i\cdot}$, $i \in \{1, \ldots, m'\}$ of the matrix A' we maximize according to $a'^{\mathrm{T}}_{i\cdot}x$ over P_{BDD}. Be γ_i the value of a corresponding longest path. If $\gamma_i \leq b'_i$ for all $i \in \{1, \ldots, m'\}$, then $P_{\mathrm{BDD}} \subseteq P'$ holds. In addition, if for all $i \in \{1, \ldots, m'\}$ $\gamma_i < b'_i$, we even have $P_{\mathrm{BDD}} \subset P'$.

4.2.4 Certificate of correctness for a threshold BDD

Given a linear constraint $a^{\mathrm{T}}x \leq b$ and a BDD via its graph representation $G = (V, A)$. The task is to check if the BDD represents the same partition of the 0/1 vertices of the unit hypercube as given by the constraint. This question arises in the context of certifying algorithms. A program is called *certifying* if it produces not only the output but also a *certificate* that this output is correct.

Our algorithms for building a threshold BDD can be extended to certifying algorithms as we show in the following. Start with building the threshold BDD for $a^{\mathrm{T}}x \leq b$. Additionally, again with the help of optimization, we give a certificate in time linear in the size of the BDD. Therefore we compute $\gamma_1 := \max\limits_{x \in \{0,1\}^d\colon x \in P_{\mathrm{BDD}}} a^{\mathrm{T}}x$ and $\gamma_0 := \min\limits_{x \in \{0,1\}^d\colon x \notin P_{\mathrm{BDD}}} a^{\mathrm{T}}x$. The value of γ_1 can be determined by the value of the longest path from the root to leaf 1 in the graph G, whereas the value of the shortest path from the root to leaf 0 in G denotes γ_0. Then (γ_1, γ_0) is the certificate. We can easily check, if $\gamma_1 \leq b < \gamma_0$ holds which means that the certificate is valid. In addition, checking if the graph G obeys the formal definition of a BDD is possible in time $\mathcal{O}(|V|)$. As we considered all 0/1 vertices of the unit hypercube, a valid certificate states, that the BDD represents the same partition of the

0/1 vertices as the given constraint. Thus the BDD corresponds to the given constraint and is correct.

4.3 0/1 Vertex counting

In this section we show how to tackle the *solution counting* problem:

Definition 4.6. *Given a set of inequalities $Ax \leq b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$, count the number of all 0/1 points satisfying the system.*

If the system $Ax \leq b$ describes a 0/1 polytope, the problem is also known as the *0/1 vertex counting* problem.

First we build the BDD for the system $Ax \leq b$. Be $G = (V, A)$ its graph representation. Then counting the 0/1 points which satisfy $Ax \leq b$ reduces to counting all vertices of the 0/1 polytope P_{BDD}. This can be achieved without explicit enumeration of all vertices in time linear in the size of the BDD as we show in the following.

W.l.o.g. be the BDD complete. With lemma 4.1 the problem is equivalent to counting the number of paths from the root to leaf 1 in G. We attach a counter $c\colon V \to \mathbb{N} \cup \{0\}$ to every node $v \in V$ which reflects the number of all paths from the root to any of the leaves that use the node v. Figure 4.2(a) shows an example.

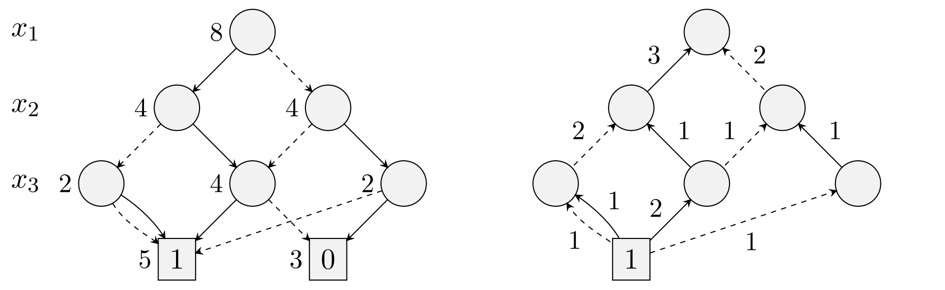
For all nodes $v \in V \setminus \{\text{root}\}$ initialize its counter $c(v) = 0$. We set $c(\text{root}) = 2^d$, which is the number of vertices of the unit hypercube and thus the total number of all paths. The idea now is to route the flow of value 2^d down through the graph and split it up equally at every node. Therefore we process all nodes $v \in V \setminus \{\text{leaf } 0, \text{leaf } 1\}$ in the descending order of their level $1, \ldots, d$. For both successor nodes w_0 and w_1 of v add half of the counter of v to their counter, i.e. $c(w_0) \leftarrow c(w_0) + \frac{c(v)}{2}$ and $c(w_1) \leftarrow c(w_1) + \frac{c(v)}{2}$. After having processed all nodes of a level labeled with x_i, $\sum_{v\colon \ell(v)=x_i} c(v) = 2^d$ holds. Finally $c(\text{leaf } 1)$ states the number of paths from the root to leaf 1 and thus the number of 0/1 vertices of the polytope P_{BDD}. In addition to that, $c(\text{leaf } 0)$ denotes the remaining number of 0/1 points not contained in P_{BDD}.

The runtime of the algorithm is $\mathcal{O}(|V|)$. Note that except for the leaves all numbers occurring in the counters are divisible by 2 since they are constructed by summation of the numbers $2^d, 2^{d-1}, \ldots, 2^2, 2^1$.

4.3.1 Center of a 0/1 polytope

A naive way to compute the center of the 0/1 polytope P_{BDD} would be to sum up all 0/1 vertices and divide the resulting vector by the total number of 0/1 vertices. This approach implies an explicit enumeration of all 0/1 vertices which can be avoided. We now come up with an extension of the vertex counting algorithm which computes the center of the 0/1 polytope P_{BDD} in time $\mathcal{O}(|V|)$.

Assume that we run the algorithm from section 4.3. Then for every nodes $v \in V$ the counter $c(v)$ reflects the total number of paths using it. Be $\nu := c(\text{leaf } 1)$ which is the number of paths from the root to leaf 1. This time the idea is to route a flow of ν units upwards from leaf 1 to the root in an appropriate way. Thereto we work on a copy



(a) Counting number of paths to leaves (b) Bottom-up flow for center of polytope

Figure 4.2: (a) For each node the number of paths using it is shown. There are 5 paths from the root to the leaf 1. (b) The center of the corresponding polytope P_{BDD} is $\left(\frac{3}{5}, \frac{2}{5}, \frac{3}{5}\right)$.

$G' = (V', A')$ of the graph G where we switch the direction of each edge and remove leaf 0. An example graph is illustrated in figure 4.2(b). We associate a flow $f_e \in \mathbb{N} \cup \{0\}$ with every edge $e \in A'$. We set the outflow of the root and the inflow of leaf 1 to ν. The inflow of leaf 0 is 0. For each node the outflow has to be the same as the inflow.

Our algorithm works as follows. We process all nodes $w \in V' \setminus \{\text{root}\}$ in ascending order of their level $d+1, d, \ldots, 2$ and start with leaf 1. For a node w be A'_w the set of outgoing edges and V'_w the set of successor nodes. For each edge $e = w \to v \in A'_w$ we put v in V'_w. Note that a node v might be contained twice in V'_w. Define $c(V'_w) := \sum_{v \in V'_w} c(v)$. Then for all edges $e = w \to v \in A'_w$ we route flow on it as $f_e = \text{inflow}(w) \cdot \frac{c(v)}{c(V'_w)}$. Note that $c(V'_w)/2$ gives the total inflow of node w in the original graph G, and $c(v)/2$ gives the flow on the incoming edge e in G. Thus, for the node w, the ratio $\frac{c(v)}{c(V'_w)}$ describes the fraction of each incoming edge on the total incoming flow in G. In G' we distribute the flow at each node w according this fraction.

Finally the flow in G' represents all paths from the root to leaf 1 in G. Therefore it reflects the addition of all 0/1 vertices of the polytope P_{BDD}. So the center x^* of P_{BDD} can be computed from the flows on the 1-edges as $x_i^* = \frac{1}{\nu} \sum\limits_{e\colon \text{head}(e)=x_i} \text{par}(e) f_e \;\; \forall i = 1, \ldots, d$.

4.4 0/1 Vertex enumeration

For a polytope there exist two independent characterizations which are equivalent, namely the representation as an $\mathcal{H}$-polytope and as a $\mathcal{V}$-polytope (we refer the reader to the preliminaries in section 2.2). Given a 0/1 polytope by a set of inequalities $Ax \le b$, we are interested in the set of all its vertices. In other words, we want to change its description from an $\mathcal{H}$-polytope to a $\mathcal{V}$-polytope. Therefore we need to solve the *0/1 vertex enumeration* problem:

Definition 4.7. *Given a set of inequalities $Ax \le b$, $A \in \mathbb{Z}^{m \times d}$, $b \in \mathbb{Z}^m$, enumerate all 0/1 points satisfying the system.*

In case $Ax \leq b$ does not describe a 0/1 polytope, the problem asks for enumerating all 0/1 solutions of the given linear system.

Bussieck and Lübbecke [BL98] presented a backtracking algorithm for solving the 0/1 vertex enumeration problem. Their algorithm intersects a polytope given by $Ax \leq b$ with two opposite facets of the unit hypercube. Then they use linear programming to check if this intersection is empty. In case it is not, it is divided into two branches which are then treated recursively. They proved that the vertex set of a 0/1 polytope is strongly $\mathcal{P}$-enumerable. For general (i.e. not necessarily 0/1) polytopes given by nondegenerate input, Avis and Fukuda [AF92] developed an algorithm which proves the strong $\mathcal{P}$-enumerability of the vertex set. If the nondegenerate requirement is dropped, it is still an open problem, whether all vertices can be enumerated in polynomial time.

Our approach for the 0/1 vertex enumeration problem is the following. We build the BDD for the system $Ax \leq b$. Then enumerating all 0/1 points which satisfy $Ax \leq b$ is equivalent to enumerating all vertices of the 0/1 polytope P_{BDD}. Be G the graph representation of the BDD. W.l.o.g. be the BDD complete. We work on a copy G' of the graph G where we switch the direction of each edge. With lemma 4.1, the enumeration of all paths from leaf 1 to the root in G' then gives all 0/1 vertices of P_{BDD}. This enumeration can be performed with a simple recursive approach. We start at leaf 1. At a node, we first process the 0-edge son and after returning from the recursion the 1-edge son. If we reach the root, we output the assignment of the x-variables corresponding to the path from the root to leaf 1 and then backtrack.

Let ν be the number of 0/1 vertices of the polytope P_{BDD}, i.e. the cardinality of the output set. The runtime for the enumeration is $\mathcal{O}(\nu d)$. Note that enumerating all 0/1 points which lie outside the polytope P_{BDD} is also possible in the same time. Solely all paths in G' starting from leaf 0 have to be enumerated.

4.5 azove – Computational results

We developed `azove` which is **a**nother **z**ero **o**ne **v**ertex **e**numeration tool. It can be downloaded from its homepage[1]. Our tool `azove` is able to count and enumerate all 0/1 solutions of a given set of linear constraints and equations, i.e. it is capable of constructing all solutions of the knapsack problem, the subset sum problem and the multidimensional knapsack problem. We provide two versions, namely `azove 1.1` and `azove 2.0`, which follow different approaches.

`azove 1.1` builds the threshold BDDs with the basic construction algorithm shown in section 3.1.1. It then sequentially uses the pairwise *and*-operation on the set of intermediate BDDs (see section 3.2.1) until one BDD is left, which is the final BDD. The order of conjunction of the BDDs can be changed with a command line option. In order to keep the size of the intermediate BDDs small, the pair of the smallest BDDs regarding the size can be chosen. If during the conjunction of the BDDs an decreasing upper bound on the number of 0/1 solutions is desired, the pair of the largest BDDs can be chosen

[1]`http://www.mpi-inf.mpg.de/~behle/azove.html`, Another Zero One Vertex Enumeration homepage, M. Behle, 2007

for conjunction. For managing the BDDs `azove 1.1` uses the `CUDD`[2] `2.4.1` library. `CUDD` automatically applies the merging and elimination to all BDDs that are constructed. In addition it provides a pairwise *and*-operator.

`azove 2.0` is based on the algorithms developed in sections 3.1.2 and 3.2.2, i.e. we implemented the output-sensitive building of threshold QOBDDs and the parallel *and*-synthesis. In contrast to version `1.1` which uses `CUDD` as BDD manager, the version `2.0` does not rely on an external library for managing BDDs. We implemented all BDD operations including the merging and elimination rule on our own. No available library for BDDs so far neither provides the datastructures which are necessary for output-sensitive building nor a parallel *and*-operation.

Both of our tools are *exact*, i.e. numerical problems are not an issue. Furthermore they are *efficient*, as the benchmarks in the following show. For building threshold BDDs they can be extended to *certifying* programs, i.e. they can give a certificate that for a linear constraint the corresponding threshold BDD was built correctly (see section 4.2.4).

0/1 Vertex counting

In this section we present computational results for counting the 0/1 solutions on a set of benchmarks with our tools `azove 1.1` and `azove 2.0`. Our benchmark set consists of different classes of problem instances, among which are SAT instances, matchings in a bipartite graph, general 0/1 polytopes and 0/1 integer linear programs.

In fields like verification and real-time system specifications counting the solutions of *SAT instances* has many applications. From several SAT competitions [BB93, HS00] we took the instances aim, hole, ca004 and hfo6. All instances are given in conjunctive normal form, i.e. a conjunction of clauses, where a clause is a disjunction of literals $\bigvee_{i\in I} x_i \vee \bigvee_{j\in J} \bar{x}_j$ with index sets $I, J \subseteq \{1,\dots,d\}$, $I \cap J = \emptyset$. Such a clause can be transformed to a linear inequality $\sum_{i\in I} x_i - \sum_{j\in J} x_j \geq 1 - |J|$ with $x_i, x_j \in \{0,1\}$ for all $i \in I$, $j \in J$. We converted all instances to sets of linear constraints and counted their satisfying solutions. The aim instances are 3-SAT instances and the hole instances encode the pigeonhole principle. There are 20 satisfiable hfo6 instances for which the results are similar. For convenience we only show the first 4 of them.

Counting the number of *matchings in a graph* is one of the most prominent counting problems with applications in physics in the field of statistical mechanics. There it arises in the study of thermodynamical properties of monomers and dimers in crystals. We counted the number of matchings for the urquhart instance, which comes from a particular family of bipartite graphs [Urq87], and for f2, which is a bipartite graph encoding a projective plane known as the Fano plane.

The two instance classes OA and TC were taken from a collection of *general 0/1 polytopes* that has been compiled in connection with [Zie95]. The convex hull of these polytopes served as input.

The problems bm23, p0033, p0040 and the stein instances are *0/1 integer linear programs* that have been taken from a collection of mixed integer linear programs, the MIPLIB [BBI92]. They are relaxations of 0/1 polytopes. Although not necessary, the

[2] `http://vlsi.colorado.edu/~fabio/CUDD`, CU Decision Diagram package homepage, F. Somenzi, 2005

Name	Dim.	Inequalities	0/1 Solutions	`azove 1.1`
OA:8-25	8	524	25	0.01
OA:9-33	9	1870	33	0.10
OA:10-44	10	9708	44	0.94
TC:8-38	8	1675	38	0.05
TC:9-48	9	6875	48	0.40
TC:10-83	10	41591	83	4.00
TC:11-106	11	250279	106	50.98
bm23	27	74	2168	3.72
p0033	33	81	10746	0.01
p0040	40	103	519216	0.01
stein15	15	66	2809	0.01
stein27	27	172	367525	0.22
stein45	45	421	244049633	232.95

Table 4.1: Counting the number of 0/1 vertices with `azove 1.1`. The runtimes are given in seconds.

bounds on the variables $0 \leq x_i \leq 1$ are given explicitly and thus included in the number of inequalities.

Table 4.1 shows the time that we need to build the BDDs and count the number of 0/1 vertices with `azove 1.1`. The tests were run on a Linux system with kernel 2.6.13 and gcc 3.4.4 on a Pentium 4 CPU with 2.6 GHz and 1.5 GB memory. We are only aware of one further tool that is capable of counting integral points in polytopes without explicit enumeration. It is `latte 1.2` [LHTY04] and implements Barvinok's algorithm [Bar94] which is polynomial in fixed dimension. A comparison with `azove 1.1` however is not fair since `latte 1.2` is not specialized in the 0/1 case. Its runtimes are considerably higher. In the following we restrict our comparisons to the two versions of `azove` since we are not aware of another tool specialized in counting 0/1 solutions for general type of problems.

In order to study the different behaviour of the sequential *and*-operation and the parallel *and*-operation, we compare `azove 1.1` with `azove 2.0`. Table 4.2 shows the comparison of the runtimes in seconds. We set a time limit of 4 hours. An asterisk marks the exceedance of this time limit. All tests were run on a Linux system with kernel 2.6.15 and gcc 3.3.5 on a 64 bit AMD Opteron CPU with 2.4 GHz and 4 GB memory. The main space consumption of `azove 2.0` is due to the storage of the signatures of the ROBDD nodes (we refer the reader to the discussion at the end of section 3.2.2). We restrict the number of stored signatures to a fixed number. In case more signatures need to be stored we start overwriting them from the beginning.

For instances with a large number of constraints `azove 2.0` clearly outperforms version `1.1`. Due to the explosion in size during the sequential *and*-operation `azove 1.1` is not able to solve some instances within the given time limit. The parallel *and*-operation in `azove 2.0` successfully overcomes this problem. On the other hand, `azove 1.1` is faster for smaller instances than version `2.0`. Since both versions incorporate approaches which are oppositional, their coexistence is justified. For a new unknown problem instance, in general both tools should be tried.

Name	Dim.	Inequalities	0/1 Solutions	azove 1.1	azove 2.0
aim-50-3_4-yes1-2	50	270	1	77.26	50.23
aim-50-6_0-yes1-1	50	400	1	43.97	9.59
aim-50-6_0-yes1-2	50	400	1	179.05	1.62
aim-50-6_0-yes1-3	50	400	1	97.24	4.58
aim-50-6_0-yes1-4	50	400	1	164.88	13.08
hole6	42	217	0	0.15	0.09
hole7	56	316	0	4.16	1.57
hole8	72	441	0	5572.74	29.69
ca004.shuffled	60	288	0	53.07	20.38
hfo6.005.1	40	1825	1	*	1399.57
hfo6.006.1	40	1825	4	*	1441.56
hfo6.008.1	40	1825	2	*	1197.91
hfo6.012.1	40	1825	1	*	1391.39
f2	49	546	151200	*	49.50
urquhart2_25.shuffled	60	280	0	*	12052.10
OA:9-33	9	1870	33	0.05	0.03
OA:10-44	10	9708	44	0.51	0.34
TC:9-48	9	6875	48	0.16	0.15
TC:10-83	10	41591	83	1.96	1.24
TC:11-106	11	250279	106	26.41	11.67

Table 4.2: Comparison of the tools `azove 1.1` and `azove 2.0`. The runtimes are given in seconds. An asterisk marks the exceedance of the time limit of 4 hours.

0/1 Vertex enumeration

The first tool that has been developed for the enumeration of 0/1 vertices is `zerone`[3]. It is based on the algorithm presented in [BL98] (for a short description of the underlying algorithm, we refer the reader to section 4.4). We compare our tool `azove 1.1` with `zerone 1.81`, which we patched to run with `CPLEX`[4] `9.0` as linear solver.

Our benchmark set contains two instance classes OA and TC, which are the convex hulls of 0/1 polytopes. Furthermore we looked at relaxations of 0/1 polytopes taken from the MIPLIB with explicitly given bounds on the variables $0 \leq x_i \leq 1$. All benchmark instances have been described in detail in the last section.

Table 4.3 shows the comparison of the runtimes in seconds. The relative amount of time spent within `azove 1.1` for the actual output of the 0/1 vertices is also given. The tests were run on a Linux system with kernel 2.6.13 and gcc 3.4.4 on a Pentium 4 CPU with 2.6 GHz and 1.5 GB memory.

For the instances p0040, stein27 and stein45 most of the time is spent for the output of the vertices. Obviously our tool `azove 1.1` outperforms `zerone 1.81` even on inequality systems that describe 0/1 polytopes and not only relaxations, in some cases by several orders of magnitude.

We also tried `vint` from the `porta`[5] `1.4.0` package which enumerates all integral points

[3]`http://www.math.tu-bs.de/mo/research/zerone.html`, zerone homepage, M. Lübbecke, 1999
[4]`http://www.ilog.com/products/cplex`, CPLEX homepage, ILOG
[5]`http://www.zib.de/Optimization/Software/Porta`, PORTA homepage, T. Christof, 2004

Name	Dim.	Inequalities	0/1 Solutions	`zerone 1.81`	`azove 1.1`	Output
OA:8-25	8	524	25	0.06	0.02	50.00 %
OA:9-33	9	1870	33	0.48	0.11	9.09 %
OA:10-44	10	9708	44	11.09	1.06	11.32 %
TC:8-38	8	1675	38	0.38	0.06	16.66 %
TC:9-48	9	6875	48	3.45	0.46	13.04 %
TC:10-83	10	41591	83	89.74	4.51	11.30 %
TC:11-106	11	250279	106	5713.19	54.53	6.51 %
bm23	27	74	2168	15.48	3.91	4.85 %
p0033	33	81	10746	11.41	0.14	92.85 %
p0040	40	103	519216	166.84	8.39	99.88 %
stein15	15	66	2809	0.41	0.02	50.00 %
stein27	27	172	367525	110.47	4.14	94.68 %
stein45	45	421	244049633	166115.17	4386.08	94.68 %

Table 4.3: Comparison of the 0/1 vertex enumeration tools `zerone 1.81` and `azove 1.1`. The runtimes are given in seconds. The Output column shows the relative amount of time spent within `azove 1.1` for the output operation of the 0/1 vertices.

in a polytope. In nearly all cases it reported that it could not handle that many inequalities. In case it succeeded the runtime was not comparable, possibly because it is not specialized in 0/1 vertices.

4.6 Facet enumeration

In this section we again use the fact that there exist two equivalent descriptions of a 0/1 polytope, namely as a $\mathcal{V}$-polytope and as an $\mathcal{H}$-polytope. This time we are interested in the transformation from $\mathcal{V}$-polytope to $\mathcal{H}$-polytope (we discussed the opposite way in section 4.4), i.e. we deal with the *facet enumeration* problem:

Definition 4.8. *Given a set $S \subseteq \{0,1\}^d$ of 0/1 points, enumerate all facets of the convex hull* $\operatorname{conv}(S)$.

This problem is also known as the *convex hull* problem. In other words, we want to compute an inequality description $P := \left\{x \in \mathbb{R}^d \mid Ax \le b,\ A \in \mathbb{Z}^{m\times d},\ b \in \mathbb{Z}^m\right\}$ such that $P = \operatorname{conv}(S)$ holds and there are no redundant inequalities. We assume in the following that the corresponding 0/1 polytope is full-dimensional. Then each inequality of the system corresponds to a facet of P. Note that every 0/1 point is a vertex of the corresponding polytope and thus there are no redundant inner points contained in the set S.

The facet enumeration problem can be converted to a vertex enumeration problem with the help of the polar S^* of S (see section 2.2). As the polar S^* may contain vertices with coefficients different from 0 and 1, we cannot use our 0/1 vertex enumeration from section 4.4. Therefore we solve this problem directly with approaches based on BDDs in the next sections.

Many algorithms have been developed for the facet enumeration resp. convex hull problem for general polytopes. In essence there are two main classes of algorithms: *incremental* algorithms and *graph traversal* algorithms.

Incremental methods successively compute a facet description of the convex hull for $S_i := \{s_1, \dots, s_i\} \subseteq S$ from the description for S_{i-1} and the additional point s_i. The first explicit description of such an algorithm now known as the *double description method* was given by Motzkin et al. [MRTT53]. Many of its ideas were refined in Kallay's (in [PS85]) and Seidel's [Sei81] *beneath and beyond* method, the randomized algorithm by Clarkson and Shor [CS89] and the derandomized version by Chazelle [Cha93]. The so-called *Fourier-Motzkin elimination* (see e.g. [Sch86]) can be viewed as a dual transformation of the double description method and thus also is an incremental method. We use the Fourier-Motzkin elimination in our approach in section 4.6.1. Bremner [Bre96] showed that incremental convex hull algorithms are not output-sensitive.

Algorithms which construct the face lattice of a polytope are called *graph traversal* methods. Among them is Chand and Kapur's *gift-wrapping* [CK70] which has been improved later by many others, e.g. by Swart [Swa85] and Rote [Rot92]. Within this algorithm, going from one facet to a neighboring one can be viewed as rotating a supporting hyperplane around the common ridge. In analogy to the 3-dimensional case, this operation is called a gift-wrapping step. Our approach in section 4.6.2 is based on gift-wrapping. In the dual sense, the main step is going from one vertex to a neighboring one. Speaking in terms of linear programming, a vertex is defined by a basis, i.e. d facets that contain the vertex. Thus exactly one member of the basis has to be exchanged, which is known as a pivoting step in the simplex algorithm. Algorithms based on this principle like Avis and Fukuda's *reverse search* [AF92] are therefore called *pivoting algorithms*.

Another approach which does neither fit in the class of incremental methods nor of graph traversal methods is the *primal-dual* method by Bremner, Fukuda and Marzetta [BFM98]. At present no polynomial runtime algorithm for the convex hull problem for general (degenerate) polytopes is known. For a more detailed overview on convex hull computations, we refer the reader to [Sei97].

4.6.1 Projection of extended flow-polytope

In section 4.1 we extended the flow-polytope (4.1) with the level equations (4.2) and got the polytope

$$P(f,x) = \{\ (f,x) \in \mathbb{R}^{m+d} \mid \begin{array}{rll} \sum\limits_{e\in\delta^+(v)} f_e - \sum\limits_{e\in\delta^-(v)} f_e = 0 & \forall\, v \in V \setminus \{\text{root}, \text{leaf } 1\} \\ \sum\limits_{e\in\delta^+(\text{root})} f_e = 1 & \\ \sum\limits_{e\in\delta^-(\text{leaf }1)} f_e = 1 & \\ x_i - \sum\limits_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e = 0 & \forall\, i \in \{1,\dots,d\} \\ f_e \geq 0 & \forall\, e \in A \\ f_e \leq 1 & \forall\, e \in A\} \end{array} \tag{4.4}$$

Then in theorem 4.1 we proved that the projection $\text{Proj}_x(P(f,x))$ of $P(f,x)$ onto the x-space is the polytope P_{BDD}. We will now compute this projection which gives us a facet description of the convex hull of P_{BDD}.

There are m flow-variables f_e and d x-variables. Note that all x-variables are implicitly bounded by $0 \le x_i \le 1$. In the following we denote the flow-variables by a number, i.e. we have $f_1, \ldots, f_m$. W.l.o.g. be P_{BDD} full-dimensional. At the beginning we use the equations to substitute flow-variables until no equations are left. Assume that m' flow-variables and k inequalities are left after these substitutions.

We now eliminate the remaining flow-variables with the Fourier-Motzkin elimination which we describe in the following. Since we may multiply inequalities by positive scalars without altering the set of solutions, we may assume that after reordering the inequalities, the remaining system is of the form

$$\begin{aligned} f_1 + \gamma_i^{\mathrm{T}} f' + a_i^{\mathrm{T}} x &\le b_i \quad \forall i \in \{1, \ldots, k'\} \\ -f_1 + \gamma_i^{\mathrm{T}} f' + a_i^{\mathrm{T}} x &\le b_i \quad \forall i \in \{k'+1, \ldots, k''\} \\ \gamma_i^{\mathrm{T}} f' + a_i^{\mathrm{T}} x &\le b_i \quad \forall i \in \{k''+1, \ldots, k\} \end{aligned}$$

with $f' := (f_2, \ldots, f_{m'})$ and appropriate vectors $\gamma_i \in \mathbb{Q}^{m'-1}$, $a_i \in \mathbb{Q}^d$ and scalars $b_i \in \mathbb{Q}$ for all $i \in \{1, \ldots, k\}$. Then the system has a solution if and only if

$$\begin{aligned} \gamma_j^{\mathrm{T}} f' + a_j^{\mathrm{T}} x - b_j \le b_i - \gamma_i^{\mathrm{T}} f' - a_i^{\mathrm{T}} x \quad & \forall i \in \{1, \ldots, k'\} \text{ and} \\ & \forall j \in \{k'+1, \ldots, k''\} \\ \gamma_i^{\mathrm{T}} f' + a_i^{\mathrm{T}} x \le b_i \quad & \forall i \in \{k''+1, \ldots, k\} \end{aligned}$$

has a solution, since an appropriate value for f_1 can be chosen. So we have projected out one flow-variable and are left with the reduced system

$$\begin{aligned} (\gamma_i + \gamma_j)^{\mathrm{T}} f' + (a_i + a_j)^{\mathrm{T}} x \le b_i + b_j \quad & \forall i \in \{1, \ldots, k'\} \text{ and} \\ & \forall j \in \{k'+1, \ldots, k''\} \\ \gamma_i^{\mathrm{T}} f' + a_i^{\mathrm{T}} x \le b_i \quad & \forall i \in \{k''+1, \ldots, k\}. \end{aligned}$$

After one iteration the number of inequalities can raise by a factor of $\lfloor \frac{k}{2} \rfloor^2$. So after some iterations the number of inequalities might have increased drastically, which is an effect known as combinatorial explosion. Many of the generated inequalities might be redundant, i.e. they can be deleted without changing the polytope that is described by the system. Therefore it is advisable to search for redundant inequalities after a few iterations and eliminate them from the system.

We iteratively eliminate all flow-variables with this method. At the end and after removal of redundant inequalities we have computed the facet description of the convex hull of P_{BDD}.

4.6.2 Gift-wrapping with a BDD

The basic outline of the gift-wrapping method is as follows: first find some initial facet of $P = \text{conv}(S)$ and the ridges that it contains. Note that a ridge r is the intersection of two facets f and f', i.e. we write $r = (f, f')$. A ridge r is called *open* if only one of its incident facets $r = (f, \cdot)$ is known. As long as there is an open ridge r, perform a gift-wrapping step to discover the other facet f' and determine the ridges that f' contains.

With this method we build the facet graph $G_P = (F, R)$ of the polytope P. The nodes $f \in F$ represent the facets of P and the edges $r \in R$ with $r = (f, f')$ represent the ridges connecting two facets. In the context of the gift-wrapping framework we have to deal with three problems (cf. [Sei97]):

(1) how to compute the ridges of the new facet f',

(2) how to maintain the set of open ridges,

(3) how to perform an individual gift-wrapping step.

Algorithm 4.1 Finding the convex hull with a BDD

```
ConvexHullBDD($S$)
1: BDD = BuildBDD($S$)
2: $f_1$ = FindFirstFacet($S$)
3: $F = \{f_1\}$
4: $R$ = FindRidges($f_1$)
5: while ($\exists$ open ridge $r = (f, \cdot) \in R$) {
6:     $f'$ = FindNewFacet($f$, $r$, BDD)
7:     $F = F \cup \{f'\}$
8:     $R = R \cup$ FindRidges($f'$) }
9: return $F$
```

Our algorithm 4.1 incorporates the BDD structure in a gift-wrapping approach. We start in step 1 with an empty BDD. For every $p \in S$ we build a BDD which represents p by its path. In analogy to section 3.2.1 we then build the synthesis of the BDDs with the pairwise *or*-operator. For the finally resulting BDD, $P_{\text{BDD}} = \text{conv}(S)$ holds. The time and space complexity for building the BDDs is naturally bounded by $\mathcal{O}(|S|d)$.

In step 2 we have to find the first facet f_1 of P to start with. Facets are represented by their normalvector and their right hand side as $a^{\mathrm{T}}x \leq b$. Be S_c the translation of the set S in such a way that $\mathbf{0} \in \text{int}(\text{conv}(S_c))$. S_c can be computed using the center of $\text{conv}(S)$ (we showed in section 4.3.1 how to determine the center of P_{BDD}). Any vertex of the polar set S_c^* of S_c can be used as the normalvector of the first facet. Such a vertex can be computed in polynomial time in $|S|$ and d. Note that it is sufficient to know the normalvector a since we can compute the right hand side b and all 0/1 vertices that lie on the facet by optimizing over P_{BDD} according to the linear objective function a. In section 4.2 we showed, that this reduces to a shortest path problem on the BDD. The value b is given as the optimal value and the shortest paths correspond to the 0/1 points that are tight at the facet.

Be S_r the set of 0/1 points that lie on a ridge r. Since a BDD implicitly represents 0/1 points in a lexicographical order, S_r is lexicographically sorted. Then the ridge r is uniquely defined by the first $d - 1$ 0/1 points that are affinely independent. This fact enables us to store ridges as bit-strings of size $(d - 1) \cdot d$.

Given a facet f we need to know all of its ridges in the steps 4 and 8. We optimize over P_{BDD} according to the normalvector of f and get the set S_f. If f is simplicial, i.e. $|S_f| = d$, all d ridges can be enumerated directly. Otherwise we compute all normalvectors a_r for all ridges of f with a sub-algorithm in dimension $d-1$ (for this purpose we can recursively use our gift-wrapping approach with a BDD or switch back to other known algorithms like e.g. lexicographical reverse search or double description method). For each normalvector a_r we calculate S_r via optimization over P_{BDD}. Solving a system of linear equations then gives us the $d-1$ affinely independent points that are the smallest regarding lexicographical order in time $\mathcal{O}\left(|S_r|^2 d\right)$.

We keep the open ridges in an additional hash-set to answer the query in step 5. The number of open ridges is bounded by the total number of facets of P. Be r a ridge which has been found by FindRidges in step 8. If r is not contained in the hash-set we add it. Otherwise we delete r from the hash-set as we know both facets that are incident to it.

Algorithm 4.2 Gift-wrapping with a BDD

```
   FindNewFacet($f$, $r$, BDD)
1: define $a' \in \mathbb{Z}^d$, $b' \in \mathbb{Z}$, $p \in \{0,1\}^d$ and $p' \in \{0,1\}^d$
2: $p$ = 0/1 point contained in $f$ but not contained in $r$
3: $a'$ = - normalvector of $f$
4: $(p', b')$ = ShortestPath($a'$, BDD)
5: while $(a'^{\mathrm{T}} p \neq b')$ {
6:     $p = p'$
7:     $a'$ = ComputeNormalvector($r$, $p$)
8:     $(p', b')$ = ShortestPath($a'$, BDD) }
9: return $(a'^{\mathrm{T}} x \leq b')$
```

The sub-routine FindNewFacet in step 6 is explained in detail in our algorithm 4.2. We use the fact that all $d-1$ points contained in the ridge r together with a 0/1 point $p \in S$ which is not contained in r define a hyperplane. We start with the given facet f and rotate it around the ridge r until a new facet f' is found. A new point p is found by optimizing over P_{BDD} in time $\mathcal{O}(|S|d)$. We have to rotate at most $|S|$ times before we find a new facet (but in practice just a few rotations are needed). Computing a normalvector of the hyperplane defined by r and p is done via solving a system of linear equations. This is possible in $\mathcal{O}\left(d^2\right)$ since r is fixed and we can do a precomputation once which costs $\mathcal{O}\left(d^3\right)$. So the overall runtime of algorithm 4.2 is $\mathcal{O}\left(d^3 + |S|(|S|d + d^2)\right)$.

If the polytope is simplicial, all steps of our algorithm 4.1 can be performed in time polynomial in d, $|S|$ and $|F|$. So in that case our algorithm is output-sensitive.

4.6.3 Computational results

In the following section we compare our implementation of the gift-wrapping approach with a BDD against other implementations for the convex hull problem which are also *exact*, i.e. which compute the results with arbitrary precision.

Neither `traf` from the `porta`[6] `1.4.0` package which implements the Fourier-Motzkin elimination nor the beneath and beyond implementation in `qhull`[7] `2003.1` compute with arbitrary precision. Therefore we did not take these programs into account.

We also implemented the Fourier-Motzkin elimination to work directly on the equations and inequalities given by the BDD structure. The corresponding polytope $P(f, x)$, which we described in section 4.6.1, is projected onto the x-space. Within our implementation we exploited the special structure of the equations. At the beginning the dependencies of the flow-variables are locally restricted within two consecutive levels. Writing the equations and inequalities levelwise these dependencies can be located in consecutive blocks of equations and inequalities. The main issue still is the rapid growth of the number of inequalities after each iteration. Although we eliminated all redundant inequalities after a few steps we could only compute the convex hull of very low dimensional polytopes, i.e. in general with dimension smaller than 6. So this approach is more of theoretical interest and not suited to work in practice. Therefore we did not take it into account in our comparison.

The runtimes of the primal-dual method implemented in `pd`[8] `1.7` [BFM98] are extremely high. The beneath and beyond implementation in `polymake`[9] `2.2` [GJ00] requires a lot of memory in higher dimensions and the runtimes are also very high. Thus we did not consider these both tools in our comparison.

Finally we restrict the comparison to `glrs`[10] `4.2` which implements the reverse search algorithm [Avi00] and `cddr+`[11] `0.77` which is an implementation of the double description method [FP96]. Both programs were compiled with the `GMP`[12] `4.1.4` for arbitrary precision.

We implemented three different versions `chBDD`, `gchBDD` and `gchBDDcdd` of the gift-wrapping approach which we developed in section 4.6.2. In the following we explain the differences between these versions. All of our implementations are exact. In our implementation `chBDD` we work with the datatype `long` until we catch an integer overflow in which case we switch to the `GMP`. This results in a speed advantage. To be able to compare with `glrs` and `cddr+` we disabled the switching of the numbertype and forced the usage of the `GMP` in our implementations `gchBDD` and `gchBDDcdd`. In the versions `chBDD` and `gchBDD` we use the implementation of the lexicographical reverse search in `lrs` as a subroutine for finding the ridges, whereas in `gchBDDcdd` we use the double description method as implemented in `cdd`. Note that our algorithm decomposes the given problems in convex hull problems in one dimension less. For these subproblems the input is lexicographically sorted. Gillmann and Kaibel [GK06] remark that this fact might help incremental methods. We also recursively used our gift-wrapping approach with a BDD as a subroutine for finding the ridges. We investigated two ways for generating the BDDs for lower dimensions, i.e. by reconstruction and by restriction of the BDD for dimension d with the help of saving extra information. In both cases the overhead slowed down the computation so that we desisted

[6] `http://www.zib.de/Optimization/Software/Porta`, PORTA homepage, T. Christof, 2004
[7] `http://www.qhull.org`, Qhull homepage, C. B. Barber and H. T. Huhdanpaa, 2003
[8] `http://www.cs.unb.ca/~bremner/pd`, pd homepage, D. Bremner, 1998
[9] `http://www.math.tu-berlin.de/polymake`, polymake homepage, E. Gawrilow and M. Joswig, 2007
[10] `http://cgm.cs.mcgill.ca/~avis/C/lrs.html`, lrslib homepage, D. Avis, 2005
[11] `http://www.ifor.math.ethz.ch/~fukuda/cdd_home`, cdd and cdd+ homepage, K. Fukuda, 2007
[12] `http://gmplib.org`, GNU Multiple Precision arithmetic library homepage, T. Granlund, 2007

from using our approach in a recursive fashion.

We run our tests on a Linux system with kernel 2.6.13 and gcc 3.4.4 on a Pentium 4 CPU with 2.6 GHz and 1.5 GB memory. All instances presented in table 4.4 were taken from a collection of 0/1 polytopes which has been compiled in connection with [Zie95], except the problem stein9 which is a 0/1 integer linear program formulation taken from the MIPLIB [BBI92]. For each instance of the benchmarks we computed the corresponding set S of satisfying 0/1 points if necessary. These sets S then served as inputs for the convex hull implementations in our comparison. The instance MJ serves as a test instance to check if the computation is really done in an exact manner. It is numerically difficult since the coefficients of the normalvectors of its facets are extremely large. In our testbed it is the only simplicial instance. All runtimes in the tables are given in seconds. The fastest runtime is printed in bold.

For the HC instances we can speed up `glrs` with our hybrid approach `chBDD`. For the higher dimensional TC problems the performance of `cddr+` can also be improved with our algorithm `chBDD`. The structures of the instances in table 4.4 do not reveal much information about the behaviour of all algorithms in general. Therefore we generated random instances as sets S of 0/1 points as follows. We generate one 0/1 point $s \in \{0,1\}^d$ by a sequence of d coin flips. If $s \notin S$ and S does not have the desired cardinality we add it in such a way that S is lexicographically sorted.

The figures 4.3, 4.4 and 4.5 show the comparisons on our randomly generated instances in dimensions 9, 10 and 11. We generated 3 instances per dimension. The figures show the median of the number of facets and the median of the runtimes. Table 4.5 gives the details. The runtimes are given in seconds and the fastest runtime is printed in bold.

The runtime of `glrs` increases with the number of vertices of the input whereas `cddr+`'s runtime mainly depends on the number of facets of the output for larger instances. The reverse search and the double description method behave complementary to each other on these instances. If the number of vertices exceeds 100 we can improve `glrs` with our hybrid approach `gchBDD` since it is easy to find all ridges in dimension $d-1$. Only for the instances with up to 300 vertices in dimension 11 we could improve `cddr+` with our hybrid approach `gchBDDcdd`. This is related to the high number of ridges that arise. For a small number of vertices our approach `chBDD` is usually the fastest. In every dimension there exists a threshold value for the number of vertices from which on our approach is inferior to `cddr+`. In dimension 9, 10 resp. 11 the crucial number of vertices lies between 200-300, 300-400 resp. 500-600. Generally speaking our approach can cope better with polytopes whose facets contain few vertices.

Name	Dim.	Vertices	Facets	`glrs`	`gchBDD`	`cddr+`	`gchBDDcdd`	`chBDD`
MJ	32	33	33	**0.01**	0.29	0.23	0.29	0.29
BIR5	16	120	25	1836.34	5576.01	**2.20**	27.55	1501.05
CUT6	15	32	368	5.67	17.29	**1.32**	28.88	4.10
HC	7	64	78	0.30	0.25	0.19	0.58	**0.07**
HC	8	128	144	4.99	3.72	0.97	2.85	**0.91**
OA	8	25	524	**0.12**	0.54	0.41	1.89	0.18
OA	9	33	1870	**0.85**	2.93	2.37	10.22	0.94
OA	10	44	9708	7.67	21.32	26.68	67.81	**6.64**
TC	7	30	432	0.11	0.31	0.35	0.95	**0.09**
TC	8	38	1675	0.54	1.71	2.14	5.22	**0.53**
TC	9	48	6875	4.87	9.98	17.18	31.10	**3.07**
TC	10	83	41591	105.57	129.03	629.12	422.46	**39.98**
TC	11	106	250279	979.97	1185.47	24532.11	6852.29	**374.44**
stein9	9	172	31	25.89	24.18	**1.57**	5.86	5.99

Table 4.4: Comparison on instances from the literature. The runtimes are given in seconds. The overall fastest runtime is printed in bold. All approaches use numbertypes from the `GMP` except from `chBDD`, which automatically switches to type `long` if possible. `gchBDD` uses `lrs` as subroutine for computing the ridges, whereas `gchBDDcdd` uses `cdd`.

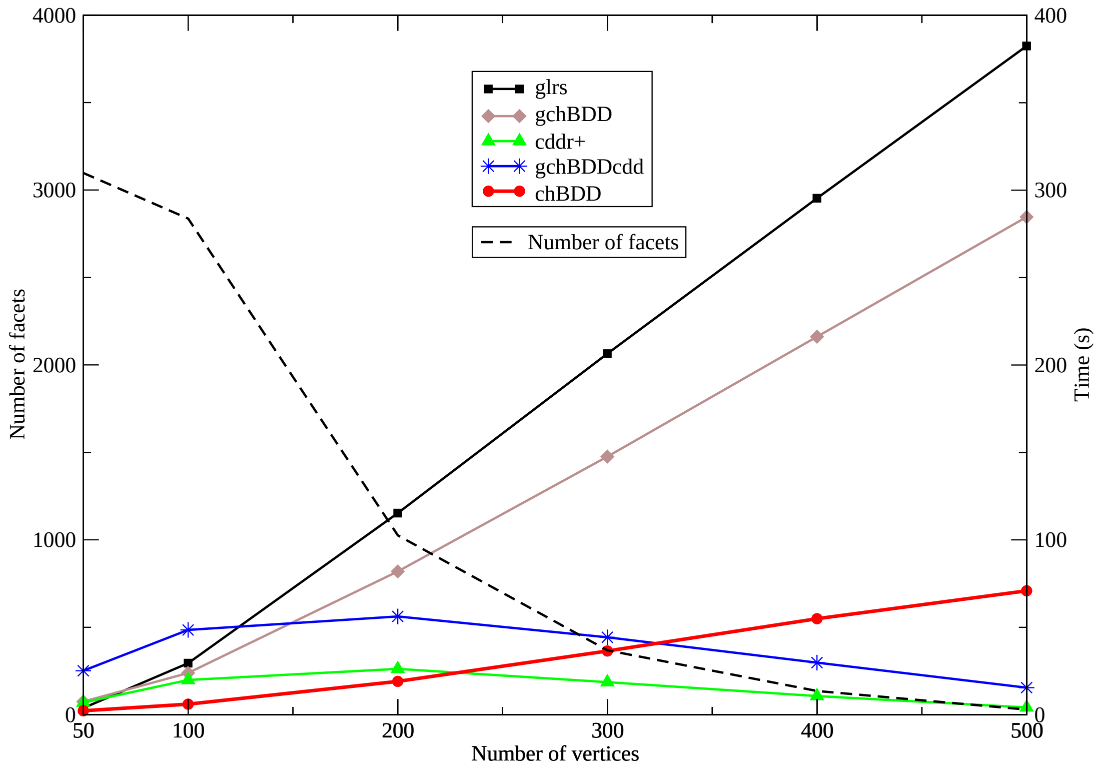


Figure 4.3: Comparison on randomly generated instances in dimension 9

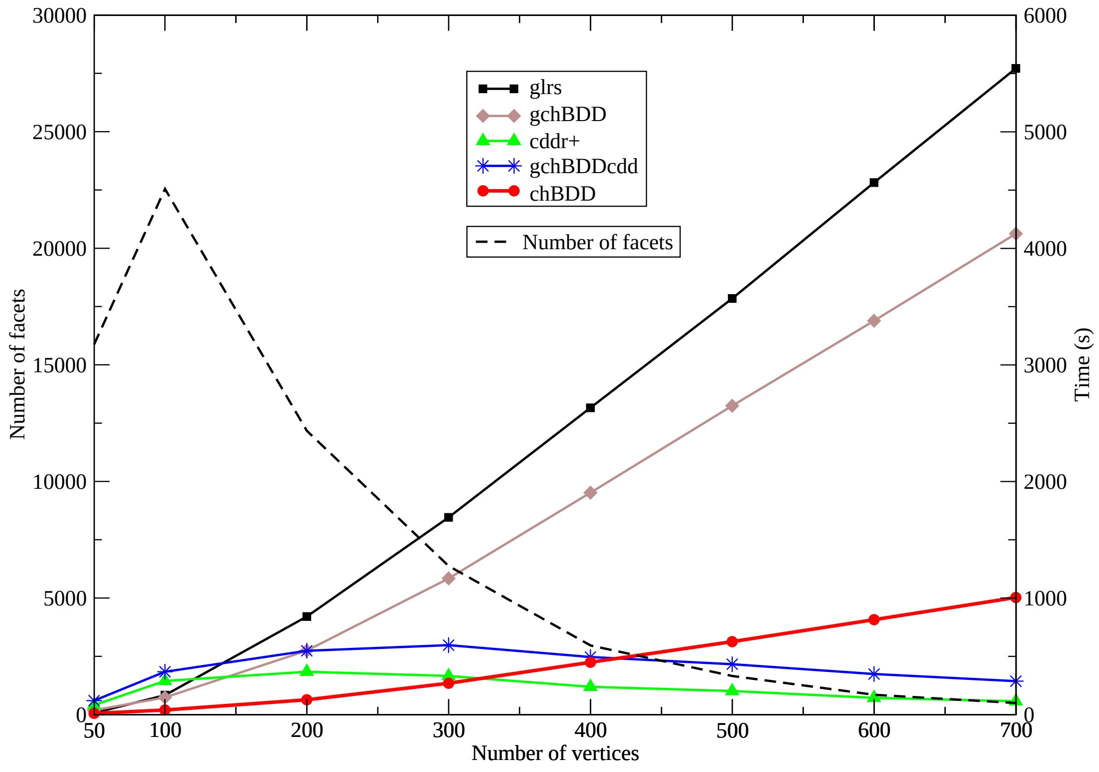


Figure 4.4: Comparison on randomly generated instances in dimension 10

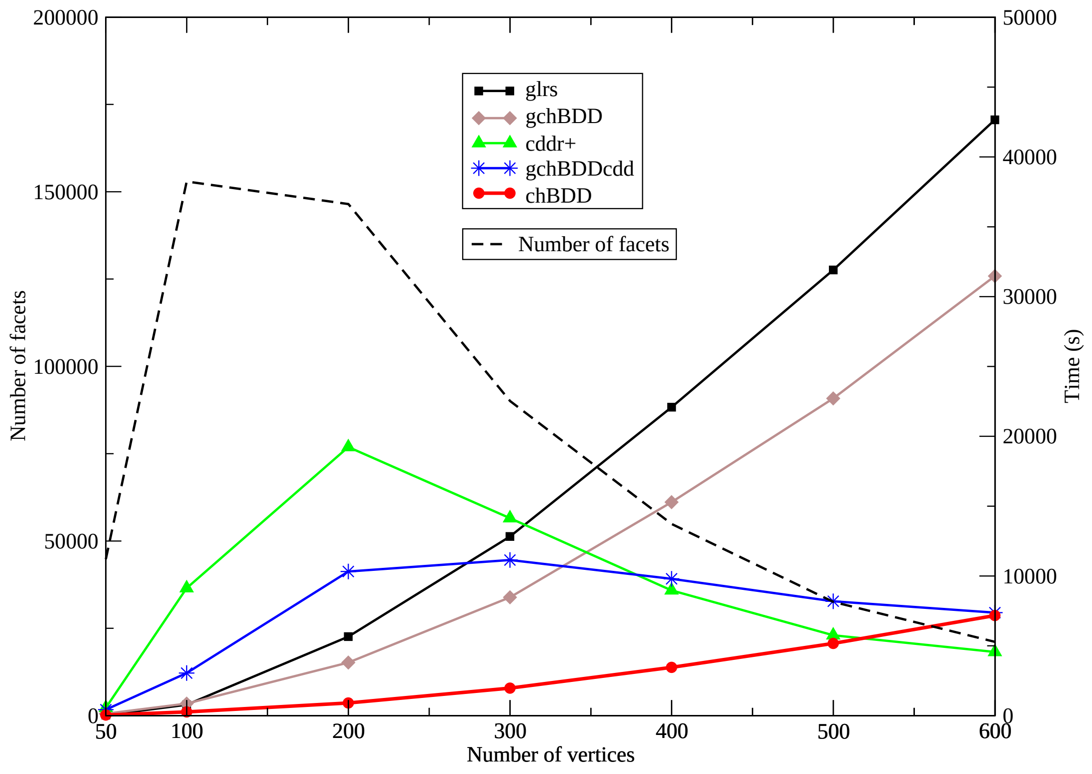


Figure 4.5: Comparison on randomly generated instances in dimension 11

Dim.	Vertices	Facets	`glrs`	`gchBDD`	`cddr+`	`gchBDDcdd`	`chBDD`
9	50	2550	3.47	6.67	5.86	22.18	**2.01**
9	50	3645	4.26	8.44	8.08	28.16	**2.53**
9	100	2519	29.16	22.67	18.46	45.42	**5.74**
9	100	3154	29.86	25.12	21.33	51.60	**6.37**
9	200	978	115.02	81.83	24.71	53.75	**19.17**
9	200	1073	115.64	82.14	27.79	58.73	**19.04**
9	300	347	209.72	147.52	**18.08**	42.88	36.44
9	300	390	203.31	147.79	**19.21**	45.74	36.53
9	400	130	293.38	215.15	**10.35**	29.00	55.21
9	400	142	297.40	217.27	**11.07**	30.64	54.62
9	500	29	380.52	285.58	**4.16**	15.42	70.63
9	500	30	384.39	283.79	**4.28**	15.60	71.19
10	50	15267	13.81	36.33	76.24	115.73	**10.89**
10	50	16488	15.07	38.63	87.25	125.42	**11.82**
10	100	21795	172.59	148.04	265.03	357.65	**39.77**
10	100	23297	161.91	147.45	314.34	378.66	**40.25**
10	200	11939	819.07	542.73	361.10	529.24	**124.50**
10	200	12401	864.17	556.29	375.52	566.54	**130.74**
10	300	6045	1700.94	1145.33	322.74	581.00	**269.83**
10	300	6697	1684.05	1191.84	340.95	611.62	**268.75**
10	400	2876	2673.48	1936.22	**231.38**	485.62	450.94
10	400	3058	2591.43	1871.79	**246.80**	503.17	447.41
10	500	1564	3578.39	2692.71	**192.43**	417.16	626.67
10	500	1752	3562.49	2607.15	**213.03**	448.77	626.24
10	600	808	4603.05	3419.75	**138.24**	340.98	823.74
10	600	895	4525.90	3338.82	**150.55**	356.17	805.52
10	700	491	5491.34	4146.82	**112.36**	281.85	1009.40
10	700	506	5597.24	4106.21	**118.15**	293.30	1001.06
11	50	44669	**40.28**	142.54	573.58	431.73	48.58
11	50	44944	**41.32**	136.22	556.49	425.34	47.24
11	100	152117	782.50	872.11	9451.48	3086.92	**266.43**
11	100	153752	781.79	860.63	8810.89	3013.82	**265.66**
11	200	142493	5631.94	3753.42	17860.01	9857.75	**905.85**
11	200	150433	5685.32	3856.81	20605.90	10784.91	**909.46**
11	300	86352	12983.14	8326.94	12965.37	10366.90	**1965.74**
11	300	93804	12672.36	8615.69	15317.20	11918.87	**1976.73**
11	400	54805	21724.94	15320.64	9761.67	10327.74	**3443.77**
11	400	54943	22445.88	15256.66	8181.38	9270.28	**3478.14**
11	500	32152	32396.92	22709.03	5595.22	8078.38	**5194.95**
11	500	32945	31422.61	22716.01	5915.44	8305.00	**5145.31**
11	600	20897	42525.57	31863.94	**4624.91**	7426.35	7140.22
11	600	21325		31089.43	**4498.19**	7322.37	7202.88

Table 4.5: Comparison on randomly generated instances in dimension 9, 10 and 11. The runtimes are given in seconds. The overall fastest runtime is printed in bold. All approaches use numbertypes from the `GMP` except from `chBDD`, which automatically switches to type `long` if possible. `gchBDD` uses `lrs` as subroutine for computing the ridges, whereas `gchBDDcdd` uses `cdd`.

5 Integer Programming

Many industrial optimization problems can be formulated as an *integer program* (IP). Formally, an integer program deals with the maximization resp. minimization of a linear objective function $c_1x_1 + \ldots + c_dx_d$, where the variables $x_1, \ldots, x_d$ have to be integers and have to satisfy m given linear inequalities $a_{i1}x_1 + \ldots + a_{id}x_d \leq b_i$ for $i \in \{1, \ldots, m\}$.

In the last decade, integer programming solvers have become one of the most important industrial strength tools to solve applied optimization problems [BFG$^+$99]. A special case of integer programming is *0/1 integer programming*, which arises if the variables are additionally restricted to attain values in $\{0, 1\}$. It is a particularly important special case, since most combinatorial optimization problems are modeled with decision variables and thus are 0/1 integer programs (0/1 IPs). In the following we will deal with 0/1 IPs, which are w.l.o.g. given in the form

$$\begin{array}{rl} \max & c^{\mathrm{T}}x \\ \text{s.t.} & Ax \leq b \\ & x \in \{0,1\}^d \end{array} \tag{5.1}$$

where $A \in \mathbb{Z}^{m\times d}$, $b \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^d$.

The most successful method for 0/1 integer programming, which is applied by all competitive commercial codes is *Branch & Cut*. This variant of Branch & Bound relies on the fact that the linear relaxation of a given 0/1 IP can be efficiently solved. The linear relaxation is the *linear program* (LP) which is obtained from the 0/1 IP by relaxing the condition $x_i \in \{0, 1\}$ to the condition $0 \leq x_i \leq 1$ for each $i \in \{1, \ldots, d\}$, that is

$$\begin{array}{rl} \max & c^{\mathrm{T}}x \\ \text{s.t.} & Ax \leq b \\ & 0 \leq x \leq 1 \end{array} \tag{5.2}$$

The value of the linear programming relaxation can then be used as an upper bound (resp. lower bound in case of minimization) in a Branch & Bound approach to solve the 0/1 IP. In Branch & Cut, one additionally applies *cutting planes* [Gom58, Sch86] to improve the quality of the linear programming relaxation. Cutting planes are inequalities which are valid for all feasible integer points, but not necessarily valid for the rational points which are feasible for the linear programming relaxation. Thus the incorporation of cutting planes improves the tightness of the linear relaxation and helps to prune parts of the Branch & Bound tree.

In theory a cutting plane can be easily inferred from a fractional optimal solution to the linear programming relaxation. The strength of the cutting plane is however crucial for the performance of the Branch & Cut process. Classes of strong valid inequalities are for example *knapsack-cover inequalities* [Bal75, CJP83, HJP75, Wol75], *clique inequalities* [NW88], the *flow-cover inequalities* [PRW85, PW84] or the *mixed integer rounding* cuts [NW88]. Knapsack-cover and flow-cover inequalities in particular are inequalities which are valid for the 0/1 points which satisfy one single constraint of the 0/1 IP.

Up to now, satisfactory methods which generate valid inequalities for the 0/1 solutions of two or more constraints are rare. The recent work [ALWW07] is an exception. Here the nature of cuts that can be constructed from two rows of the simplex tableau is investigated. In this chapter we also aim at a method for the algorithmic problem of generation of valid inequalities from multiple constraints. Our approach is based on *Binary Decision Diagrams* which gives the flexibility to choose any subset of the constraints given in the 0/1 IP for generation of cuts.

Currently there is active and promising research in the field of combining techniques from computational logic and constraint programming with integer programming, see e.g. [CF04, Hoo04]. We contribute further to this development by using BDDs successfully and for the first time in a cutting plane engine resp. Branch & Cut framework.

Lai et al. [LPV93, LPV94] have developed a Branch & Bound algorithm for solving 0/1 IPs that uses an extension of BDDs called *Edge Valued Binary Decision Diagrams (EVBDDs).* EVBDDs represent functions $f\colon \{0,1\}^d \to \mathbb{Z}$. So the EVBDDs are used not only to represent the characteristic functions of the constraints but also for the constraints themselves. In their approach however, one has to build an EVBDD for the conjunction of all the constraints of the given 0/1 IP. In many cases this leads to an explosion in memory requirement.

In section 5.1 we discuss our method of how to build and apply a BDD in a node of the Branch & Cut tree. We use BDDs to represent the feasible 0/1 solutions of a subset $A'x \leq b'$ of the given constraints $Ax \leq b$. Specific to our approach is, that we can choose any subset of constraints, i.e. it is possible to generate cuts from one, two or multiple constraints. In addition we do not need to build the BDD for all constraints, and thereby avoid the explosion of the size of the BDD which might happen if the BDD has to be built for all the constraints in $Ax \leq b$.

The central part of this chapter forms section 5.2. Here we show how to derive valid inequalities for the polytope which is described by the 0/1 solutions of the constraint set $A'x \leq b'$, which are represented by a BDD. We use BDDs in several approaches for the separation of cutting planes, among which are linear programming and Lagrangean relaxation in combination with the subgradient method.

In order to be able to integrate these procedures into a Branch & Cut framework for 0/1 IPs, we also need strengthening of the separated cuts, which we develop in section 5.3, and lifting, which we discuss in section 5.4.

Finally in section 5.5 we describe the details of our implementation of the developed techniques and give computational results. We incorporate BDDs into a cutting plane engine and apply it in a Branch & Cut framework of an integer programming solver. The separation problem is solved with a sequence of shortest path problems with Lagrangean relaxation techniques. For this we use a standard BDD-package and apply our own efficient implementation of an acyclic shortest path algorithm on the BDD-datastructure. We

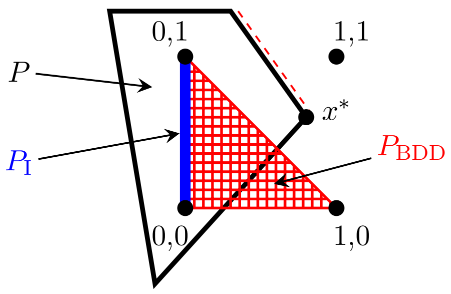


Figure 5.1: A 2-dimensional example showing the relation between the polytopes P, P_{I} and P_{BDD}. The BDD corresponding to P_{BDD} was built for the dashed constraint.

applied our cutting plane framework to the *MAX-ONES* problem and to *randomly generated* 0/1 IPs. Our computational results show that we could develop competitive code to solve hard 0/1 integer programming problems, on which state-of-the-art commercial Branch & Cut codes fall short.

5.1 Using BDDs in Branch & Cut

Suppose we have to solve a 0/1 integer programming problem of the form (5.1) and use Branch & Cut for this task. In a node of the Branch & Cut tree we then have the set of constraints $Ax \leq b$ and, in case we are not in the root node, some variables might be fixed to the values 0 or 1. That is, we have the decomposition of the index set $\{1, \ldots, d\}$ into three disjoint sets $I_0 \,\dot\cup\, I_1 \,\dot\cup\, F = \{1, \ldots, d\}$. The fixation of variables is then given by $\forall i \in I_0 : \ x_i = 0$ and $\forall i \in I_1 : \ x_i = 1$. In the following we implicitly assume that the variables are fixed according to the sets I_0 and I_1.

The polytope $P := \{x \in [0,1]^d \mid Ax \leq b\}$ is naturally defined by the LP relaxation of the problem. We are interested in the polytope $P_{\mathrm{I}} := \mathrm{conv}(x \in \{0,1\}^d \mid Ax \leq b)$, which is the convex hull of the feasible 0/1 points contained in P. Our idea is now to choose a subset $A'x \leq b'$ of the constraints in $Ax \leq b$ and to build the BDD which represents all 0/1 points which satisfy $A'x \leq b'$. We next distinguish between the two 0/1 polytopes P_{I} and P_{BDD}. In this case the polytope $P_{\mathrm{BDD}} := \mathrm{conv}(x \in \{0,1\}^d \mid A'x \leq b')$ is the convex hull of the 0/1 points which are feasible for $A'x \leq b'$. Clearly $P_{\mathrm{BDD}} \supseteq P_{\mathrm{I}}$, i.e. P_{BDD} is an overapproximation of P_{I}. Figure 5.1 illustrates how the three polytopes P, P_{I} and P_{BDD} are related.

In our current node of the Branch & Cut tree, be $x^* \in [0,1]^d$ the optimal solution of the LP relaxation. We choose the constraints for the subset $A'x \leq b'$, which define P_{BDD}, from those constraints in $Ax \leq b$ that are tight at x^*. Ideally we are able to build the BDD for a maximum subset of independent constraints, as e.g. the basis of x^*. If we could build the BDD for all constraints in $Ax \leq b$, we would be done because then $P_{\mathrm{BDD}} = P_{\mathrm{I}}$ holds. In that case we could solve the 0/1 integer programming problem with the optimization method presented in section 4.2. In practice building the BDD for all constraints in $Ax \leq b$

usually takes more time than solving the 0/1 IP with a Branch & Cut approach. However, if there is a sequence of 0/1 IPs which differ only in the objective function, there is a break-even, i.e. if the number of 0/1 IPs is large enough, it is benefiting to build the complete BDD once instead of solving all 0/1 IPs independently with Branch & Cut.

5.1.1 Learning

In the area of *SAT solving*, the main task is to determine the *satisfiability* of a boolean formula, which is mostly given in conjunctive normal form (CNF). There is a close relation between SAT problems and 0/1 IPs, i.e. each SAT problem can also be modelled as a 0/1 IP. Many techniques for the transformation of a CNF into a 0/1 IP are known (see e.g. [CPS90]).

One of today's main methods for solving SAT problems is the DPLL algorithm, named after Davis, Putnam, Logemann, and Loveland. It employs a systematic backtracking procedure to explore the space of variable assignments, searching for a satisfying assignment. In modern SAT solvers, the basic search procedure is augmented by clause learning. In principle the Branch & Cut method does something similar for solving 0/1 IPs. The backtracking procedure is realized with the Branch & Bound tree, whereas new constraints are "learned" by separation of cutting planes. In this context feasibility is determined as a byproduct. Here is a simple example. Given the following CNF

$$\begin{array}{ll} & (\bar{x}_1 \vee x_2 \vee x_3) \\ \wedge & (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \end{array}$$

then via resolution, the clause

$$\bar{x}_1 \vee x_3$$

can be learned. For the corresponding 0/1 IP

$$\begin{aligned} -x_1 + x_2 + x_3 &\geq 0 \\ -x_1 - x_2 + x_3 &\geq -1 \end{aligned}$$

in an analogous manner the constraint

$$-x_1 + x_3 \geq 0$$

can be derived.

In the following we will present two classes of cuts that express logical consequences, which can be derived from building the BDD.

Exclusion cut

In the current node of the Branch & Cut tree, let some variables be fixed, i.e. we have $\emptyset \neq F \neq \{1, \ldots, d\}$. If we have built the BDD according to this fixation and detect, that it only consists of a long edge from the root to the leaf 0, we can conclude that there are no feasible binary points in P with variables set as given by I_0 and I_1. Therefore we exclude these kinds of binary points by an exclusion cut.

Consider the linear function $\sum_{i\in I_1} x_i + \sum_{i\in I_0}(1-x_i)$. All points that we want to exclude maximize this function and the maximum value is $|I_1| + |I_0|$. To cut off these points we subtract 1 from the maximum value. This leads to the *exclusion cut*

$$\begin{aligned} \sum_{i\in I_1} x_i + \sum_{i\in I_0}(1-x_i) \le |I_1| + |I_0| - 1 \\ \Leftrightarrow \sum_{i\in I_1} x_i - \sum_{i\in I_0} x_i \le |I_1| - 1 \end{aligned}$$

By construction it is valid for all feasible 0/1 points in P.

Implication cut

Consider again the decomposition of the index set $\{1,\dots,d\} = I_0 \dot\cup I_1 \dot\cup F$ and let some variables be fixed, i.e. $\emptyset \neq F \neq \{1,\dots,n\}$. Again we build the BDD corresponding to the fixation given by $\forall i \in I_0 : \; x_i = 0$ and $\forall i \in I_1 : \; x_i = 1$. If there are variables, for whom in the graph representation of the BDD all outgoing 0-edges resp. 1-edges lead to leaf 0, then the BDD-polytope P_{BDD} is not full-dimensional. In such a situation the fixation of variables according to I_0 and I_1 implicates the fixation of other variables which can be expressed by implication cuts.

Assume that all 1-edges for x_j with $j \in F$ lead to leaf 0. Then we have the following logical implication

$$\bigwedge_{i\in I_1} x_i \wedge \bigwedge_{i\in I_0} \bar{x}_i \rightarrow \bar{x}_j \;\equiv\; \bigvee_{i\in I_1} \bar{x}_i \vee \bigvee_{i\in I_0} x_i \vee \bar{x}_j$$

This can be expressed as an *implication cut*

$$\begin{aligned} \sum_{i\in I_1}(1-x_i) + \sum_{i\in I_0} x_i + (1-x_j) \ge 1 \\ \Leftrightarrow \sum_{i\in I_1} x_i - \sum_{i\in I_0} x_i + x_j \le |I_1| \end{aligned}$$

Now assume that all 0-edges for x_j lead to leaf 0. This leads to the logical implication

$$\bigwedge_{i\in I_1} x_i \wedge \bigwedge_{i\in I_0} \bar{x}_i \rightarrow x_j \;\equiv\; \bigvee_{i\in I_1} \bar{x}_i \vee \bigvee_{i\in I_0} x_i \vee x_j$$

Now the corresponding implication cut is

$$\begin{aligned} \sum_{i\in I_1}(1-x_i) + \sum_{i\in I_0} x_i + x_j \ge 1 \\ \Leftrightarrow \sum_{i\in I_1} x_i - \sum_{i\in I_0} x_i - x_j \le |I_1| - 1 \end{aligned}$$

All cuts are by construction valid for all feasible 0/1 points in P.

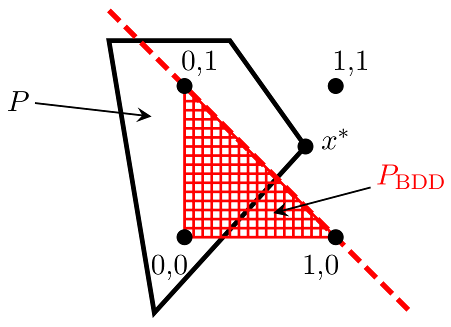


Figure 5.2: The dashed line represents a cutting plane, which separates x^* from P_{BDD}. In this case, its intersection with P_{BDD} defines a facet of P_{BDD}.

5.2 Separation with BDDs

In a Branch & Cut framework, we want to decide in each node of the Branch & Cut tree, whether our current optimal solution x^* to the linear programming relaxation P also lies in the convex hull of the integral points, i.e. if $x^* \in P_{\mathrm{I}}$ holds. If not, we want to strengthen the description of the relaxation by adding an inequality to it, which is valid for all points in P_{I} but not valid for x^*. This is the so-called *separation problem.*

Definition 5.1 (Separation Problem)**.** *Given $x^* \in \mathbb{R}^d$, find an inequality, which is valid for P_{I} and violated by x^*, or assert that such does not exist.*

In case such an inequality exists, its corresponding equality defines a hyperplane which is called *cutting plane* or *separating hyperplane.*

Now recall, that independent from the set of constraints $A'x \leq b'$ which we have chosen to build the BDD for, P_{BDD} is at most an overapproximation of P_{I}. In the following we use this fact and extend the separation problem to our context.

Definition 5.2 (BDD-SEP)**.** *Given $x^* \in \mathbb{R}^d$ and a BDD for which $P_{\mathrm{I}} \subseteq P_{\mathrm{BDD}}$ holds, find an inequality, which is valid for P_{BDD} and violated by x^*, or assert that such does not exist.*

Figure 5.2 depicts an example of a cutting plane whose corresponding inequality is valid for P_{BDD} and thus also for P_{I}.

5.2.1 Polynomial time solvability of BDD-SEP

In the beginning of the 1980's, several authors [GLS81, KP80, PR81] showed that the linear optimization problem over polyhedra and the separation problem over polyhedra are polynomial time equivalent. The equivalence of separation and optimization is established via the ellipsoid method and the γ-polar of a polyhedron, see [GLSv88].

This equivalence of separation and optimization is a central result in combinatorial optimization. In our context, it implies that one can solve the separation problem for P_{BDD} (BDD-SEP, see definition 5.2) in polynomial time, if one can solve the optimization problem for P_{BDD} (BDD-OPT, see definition 4.3) in polynomial time. The later is provided in section 4.2 by theorem 4.2, thus we can conclude that BDD-SEP can, in theory, also be efficiently solved.

Theorem 5.1. *BDD-SEP can be solved in polynomial time.*

5.2.2 Separation via solving a Linear Program

In section 4.1 we developed the polytope $P(f,x)$ as an extension of the flow-polytope (4.1) of the graph representation of the BDD together with the level equations (4.2). An explicit description of $P(f,x)$ is given in section 4.6.1 by (4.4) on page 43. We now use this formulation of the BDD in the space $\mathbb{R}^{m+d}$ to solve the separation problem BDD-SEP with one call to a linear programming algorithm.

Consider the explicit description (4.4) of the polytope $P(f,x)$. Let the matrices C and D and the vector h be according to this description, so that we can alternatively write

$$P(f,x) = \{(f,x) \in \mathbb{R}^{m+d} \mid Cf + Dx \leq h\}.$$

In association with the projection of $P(f,x)$ onto the x-space $\text{Proj}_x(P(f,x))$, define the *projection cone* as

$$W := \{v \in \mathbb{R}^k \mid v^{\mathrm{T}}C = 0,\ v \geq 0\}.$$

Now the projection $\text{Proj}_x(P(f,x))$ can also be written (see [Bal01]) as

$$\text{Proj}_x(P(f,x)) = \{x \in \mathbb{R}^d \mid (v^{\mathrm{T}}D)x \leq v^{\mathrm{T}}h,\ v \in \text{extr}(W)\} \tag{5.3}$$

where $\text{extr}(W)$ denotes the set of extreme rays of W. Finally, our theorem 4.1 provides

$$P_{\text{BDD}} = \text{Proj}_x(P(f,x)).$$

Using the description (5.3) the separation problem BDD-SEP can then be solved as follows. We rewrite

$$\begin{array}{rrcl} & v^{\mathrm{T}}Dx^* & \leq & v^{\mathrm{T}}h \\ \Leftrightarrow & v^{\mathrm{T}}(Dx^* - h) & \leq & 0 \end{array}$$

Solving BDD-SEP then reduces to solving the linear program

$$\begin{array}{rl} \max & (Dx^* - h)^{\mathrm{T}}v \\ \text{s.t.} & v^{\mathrm{T}}C = 0 \\ & v \geq 0 \end{array}$$

If $x^* \notin P_{\text{BDD}}$ the above LP might be unbounded. W.l.o.g. we therefore normalize v by adding the linear constraint $\|v\|_1 \leq 1$ to the LP. Let $\tilde{v}$ be the optimal solution to the LP. If $(Dx^* - h)^{\mathrm{T}}\tilde{v} > 0$ then $\tilde{v}^{\mathrm{T}}Dx \leq \tilde{v}^{\mathrm{T}}h$ is an inequality valid for P_{BDD} and violated by x^*. Otherwise we have $x^* \in P_{\text{BDD}}$.

Simplification of the separation LP

We now take a closer look at the representation (4.4) of the polytope $P(f,x)$ to define the matrices C and D and the vector h explicitly. We have

	Cf	$+Dx$	$\leq h$
$\forall v \in V$	$\sum\limits_{e\in\delta^-(v)} f_e - \sum\limits_{e\in\delta^+(v)} f_e$		$= \begin{cases} -1 & \text{if } v = \text{root} \\ 1 & \text{if } v = \text{leaf } 1 \\ 0 & \text{otherwise} \end{cases}$
$\forall e \in A$	$-f_e$		≤ 0
$\forall e \in A$	f_e		≤ 1
$\forall i \in \{1,\dots,d\}$	$\sum\limits_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e$	$-x_i$	$= 0$

The outflow of the root node is 1. Therefore we can drop the inequalities $\forall e \in A:\ f_e \leq 1$. We see that matrix C consists of three type of matrices namely a node-edge incidence matrix which we call N, a negative identity matrix of dimension $|A|$ called $-I_{|A|}$, and a level-parity-one-edge incidence matrix called L. Matrix D consists of a $(|V|+|A|)\times d$ dimensional 0-matrix and $-I_d$. The vector h only has two nonzero entries within the first $|V|$ entries. So it can be written as $h^{\mathrm{T}} = (h_{|V|}^{\mathrm{T}}, 0_{|A|}^{\mathrm{T}}, 0_d^{\mathrm{T}})$. We split up the vector $v^{\mathrm{T}} = (v_{|V|}^{\mathrm{T}}, v_{|A|}^{\mathrm{T}}, v_d^{\mathrm{T}})$. This leads to the following separation LP

$$\begin{array}{ll} \max & -h_{|V|}^{\mathrm{T}} v_{|V|} - (x^*)^{\mathrm{T}} v_d \\ \text{s.t.} & \begin{pmatrix} N^{\mathrm{T}}, & -I_{|A|}, & L^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} v_{|V|} \\ v_{|A|} \\ v_d \end{pmatrix} = 0 \\ & v_{|V|} \in \mathbb{R}^{|V|} \\ & v_{|A|} \geq 0 \\ & v_d \in \mathbb{R}^d \end{array}$$

The variables $v_{|V|}$ and v_d are now continuous because they arise from dualization of equations. We can write $-h_{|V|}^{\mathrm{T}} v_{|V|}$ explicitly as $v_{\{\text{root}\}} - v_{\{\text{leaf } 1\}}$ where $v_{\{\text{root}\}}$ and $v_{\{\text{leaf } 1\}}$ are the entries of $v_{|V|}$ which correspond to the root and the leaf 1 node. The variables $v_{|A|}$ do not appear in the objective function. In fact they act as slack variables and can therefore be eliminated. So our new simplified separation LP is

$$\begin{array}{ll} \max & v_{\{\text{root}\}} - v_{\{\text{leaf } 1\}} - (x^*)^{\mathrm{T}} v_d \\ \text{s.t.} & \begin{pmatrix} N^{\mathrm{T}}, & L^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} v_{|V|} \\ v_d \end{pmatrix} \geq 0 \\ & v_{|V|} \in \mathbb{R}^{|V|} \\ & v_d \in \mathbb{R}^d \end{array} \tag{5.4}$$

In case $x^* \notin P_{\mathrm{BDD}}$ the above LP might again be unbounded. This time we guarantee the existence of a finite optimum by bounding v via $\|v\|_\infty \leq 1$. Let $\tilde{v}$ again be the optimal solution to the LP. If the value of the maximum is > 0 then $\tilde{v}_d^{\mathrm{T}} x \geq \tilde{v}_{\{\mathrm{root}\}} - \tilde{v}_{\{\mathrm{leaf}\ 1\}}$ is an inequality which is valid for P_{BDD} and violated by x^*, and otherwise $x^* \in P_{\mathrm{BDD}}$ holds.

5.2.3 A cutting plane approach for BDD-SEP

In section 5.2.1 we have referred to the classical result of equivalence of separation and optimization [GLSv88], which is established via the ellipsoid method and the γ-polar of a polyhedron. Applied to BDD-SEP, one can establish a linear program to solve this separation problem, which has an exponential number of constraints. These constraints in turn, can be separated with optimization over P_{BDD}, i.e. solving BDD-OPT, which we have treated in section 4.2. So the number of constraints to be separated is bounded by the number of 0/1 vertices of P_{BDD}.

Recall that the γ-polar for P_{BDD} consists of all $y \in \mathbb{R}^d$ and $\gamma \in \mathbb{R}$, such that

$$y^{\mathrm{T}} x \leq \gamma \tag{5.5}$$

holds for all $x \in P_{\mathrm{BDD}}$. For every y, we can compute the according γ as

$$\gamma_y := \max\{y^{\mathrm{T}} x \mid x \in P_{\mathrm{BDD}}\}, \tag{5.6}$$

so that the inequality $y^{\mathrm{T}} x \leq \gamma_y$ is valid for P_{BDD}.

Assume now, that we are given a vector $x^* \in \mathbb{R}^d$ and want to solve BDD-SEP. Among the inequalities (5.5), we are looking for one which additionally fulfills

$$y^{\mathrm{T}} x^* > \gamma.$$

If such a y exists, then, by scaling y appropriately, there exists a y with

$$y^{\mathrm{T}} x^* \geq \gamma_y + 1. \tag{5.7}$$

Following ideas similar to the ones in [BCC93], we aim at finding a y of minimal $\|\cdot\|_1$-norm which satisfies (5.7). Therefore we substitute $|y_i|$ by $h_i \in \mathbb{R}_{\geq 0}$ and additional constraints $-h_i \leq y_i \leq h_i$. The solution of the following linear program gives us the desired y.

$$\begin{array}{lrcll} \min & \sum_{i=1}^{d} h_i & & & \\ \text{s.t.} & y^{\mathrm{T}} x^* & \geq & y^{\mathrm{T}} x + 1 & \forall x \in P_{\mathrm{BDD}} \qquad (*) \\ & y_i & \leq & h_i & \forall i \in \{1, \ldots, d\} \\ & -y_i & \leq & h_i & \forall i \in \{1, \ldots, d\} \\ & y \in \mathbb{R}^d & & & \\ & h \in \mathbb{R}^d_{\geq 0} & & & \end{array} \tag{5.8}$$

The separation problem for the set of constraints $(*)$ in (5.8) can be solved via optimizing over P_{BDD}. In this way, one can implement a *cutting plane approach* [DFJ54] to solve the linear program (5.8) as follows. We start with a LP of type (5.8) with a nonempty, preferably small set of constraints $(*)$. If the LP is infeasible, we have $x^* \in P_{\mathrm{BDD}}$. Otherwise, solving this LP gives us a candidate y. If optimizing over P_{BDD} according to this

y finds a violated constraint of type $(*)$, we add it to the LP and iterate again, until the LP turns infeasible or no violated constraint can be found. In the latter case, the final solution y, together with γ_y computed as in (5.6), gives the inequality $y^{\mathrm{T}}x \leq \gamma_y$ which solves BDD-SEP.

5.2.4 Separation via Lagrangean relaxation and the subgradient method

Our aim now is to solve the separation problem BDD-SEP for any given $x^* \in \mathbb{R}^d$ without solving a linear program. This can be achieved via a combination of the Lagrangean relaxation and the subgradient method (see e.g. [Sch86]).

Recall the way we constructed the polytope $P(f, x)$ in section 4.1 by adding the level equations (4.2) to the flow-polytope P_{flow} (4.1). If $x^* \in P_{\text{BDD}}$, there exists a flow $f \in P_{\text{flow}}$ representing x^*, so there exists a solution to the following linear problem

$$\begin{array}{ll} \min & 0 \\ \text{s.t.} & f \in P_{\text{flow}} \\ & \displaystyle\sum_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e = x_i^* \quad \forall i \in \{1, \ldots, d\} \end{array}$$

Considering this trivial minimization problem, the level equations (4.2) are the constraints which make the problem hard to solve. If we remove them from the list of constraints and put them into the objective function, we obtain a lower bound

$$0 \geq \max_{\lambda \in \mathbb{R}^d} \underbrace{\Big(\lambda^{\mathrm{T}} x^* + \min_{f \in P_{\text{flow}}} \sum_{i=1}^{d} -\lambda_i \sum_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e \Big)}_{:=\text{LR}(\lambda)}$$

The maximum here is called the *Lagrangean relaxation* (LR) and the components of λ the *Lagrange multipliers*. The inequality can be shown easily: be $x^* \in P_{\text{BDD}}$ and $f^* \in P_{\text{flow}}$ the appropriate flow, and λ' attains the maximum. Then

$$0 = \lambda'^{\mathrm{T}} x^* + \Big(\sum_{i=1}^{d} -\lambda'_i \sum_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e^* \Big) \geq \text{LR}(\lambda')$$

For a given λ, finding the minimum of $\displaystyle\min_{f \in P_{\text{flow}}} \sum_{i=1}^{d} -\lambda_i \sum_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e$ reduces to a minimization problem over P_{BDD} with the linear objective function $-\lambda \in \mathbb{R}^d$. In section 4.2 we have seen, that we can solve such kind of optimization problem (BDD-OPT) in linear time in the size of the BDD. We define

$$\delta_\lambda := \max_{f \in P_{\text{flow}}} \sum_{i=1}^{d} \lambda_i \sum_{\substack{e:\text{head}(e)=x_i \\ \text{par}(e)=1}} f_e$$

so that we can write

$$\text{LR}(\lambda) = \lambda^{\text{T}} x - \delta_\lambda.$$

Suppose now we have to solve BDD-SEP for $x^* \in \mathbb{R}^d$. The point $x^* \notin P_{\text{BDD}}$ if and only if there exists a $\lambda \in \mathbb{R}^d$ such that $\text{LR}(\lambda) > 0$, i.e.

$$\lambda^{\text{T}} x^* > \delta_\lambda.$$

In order to find such a λ, we use the *subgradient method* as given in Alg. 5.1.

Algorithm 5.1 Subgradient method for BDD-SEP

```
1: $k := 1$
2: $\lambda^{(k)} := c \in \mathbb{R}^d \setminus \{\mathbf{0}\}$
3: Compute a longest path $x_{p^{(k)}}$ from root to leaf 1 w.r.t. $\lambda^{(k)}$ with length $\delta_{\lambda^{(k)}}$
4: **if** ${\lambda^{(k)}}^{\text{T}} x^* > \delta_{\lambda^{(k)}}$ **then return** the inequality ${\lambda^{(k)}}^{\text{T}} x \le \delta_{\lambda^{(k)}}$
5: $t^{(k)} := \frac{1}{k}$
6: $\lambda^{(k+1)} := \lambda^{(k)} + t^{(k)}(x^* - x_{p^{(k)}})$
7: $\text{k} := \text{k} + 1$
8: **goto** 3
```

In words, Alg. 5.1 does the following. We start with $\lambda^{(1)} \in \mathbb{R}^d \setminus \{\mathbf{0}\}$. In case of performing the separation in a Branch & Cut framework, the first guess for a normalvector of a separating hyperplane is the objective function vector c, which is why λ is initialized with this vector. Let $\lambda^{(k)}$ be the normalvector in the k-th iteration, and be $x_{p^{(k)}}$ the longest path in the BDD from root to leaf 1 w.r.t. the length function $\lambda^{(k)}$. Its length is given by $\delta_{\lambda^{(k)}} = {\lambda^{(k)}}^{\text{T}} x_{p^{(k)}}$. If ${\lambda^{(k)}}^{\text{T}} x^* > \delta_{\lambda^{(k)}}$ we have found the inequality ${\lambda^{(k)}}^{\text{T}} x \le \delta_{\lambda^{(k)}}$, which is valid for P_{BDD} and separates x^* from P_{BDD} and thus solves BDD-SEP. Assume now that still ${\lambda^{(k)}}^{\text{T}} x^* \le \delta_{\lambda^{(k)}}$ holds. After the update one has

$$\begin{aligned} {\lambda^{(k+1)}}^{\text{T}}(x^* - x_{p^{(k)}}) &= {\lambda^{(k)}}^{\text{T}}(x^* - x_{p^{(k)}}) + t^{(k)}\|x^* - x_{p^{(k)}}\|^2 \\ \Leftrightarrow \quad {\lambda^{(k+1)}}^{\text{T}} x^* - {\lambda^{(k+1)}}^{\text{T}} x_{p^{(k)}} - t^{(k)}\|x^* - x_{p^{(k)}}\|^2 &= {\lambda^{(k)}}^{\text{T}} x^* - \delta_{\lambda^{(k)}} \end{aligned}$$

If the step length $t^{(k)} > 0$ is small enough then there exists a longest path $x_{p^{(k)}}$ w.r.t. $\lambda^{(k)}$, which is also a longest path w.r.t. $\lambda^{(k+1)}$. Then we have $\delta_{\lambda^{(k+1)}} = {\lambda^{(k+1)}}^{\text{T}} x_{p^{(k)}}$ and thus

$${\lambda^{(k+1)}}^{\text{T}} x^* - \delta_{\lambda^{(k+1)}} > {\lambda^{(k)}}^{\text{T}} x^* - \delta_{\lambda^{(k)}}.$$

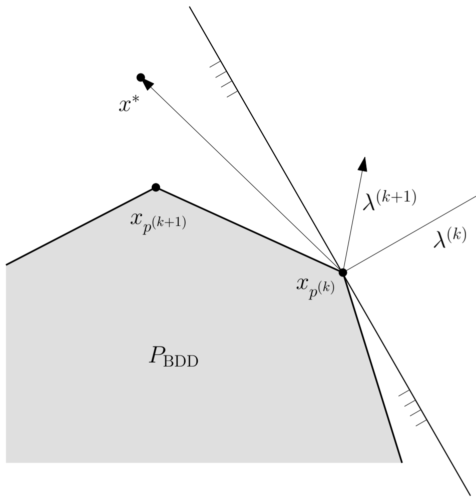


Figure 5.3: Illustration of the $(k+1)$-th iteration of the subgradient method. After this iteration, the separating hyperplane for x^* will be found.

This means that we have an increase of the value of the Lagrangean relaxation after each iteration with a suitable $t^{(k)}$. It is known (see e.g. [Sch86]), that for any $t^{(k)}$ with $\lim_{k\to\infty} t^{(k)} = 0$ and $\sum_{k=1}^{\infty} t^{(k)} = \infty$ the subgradient method terminates. This is the case for $t^{(k)} = \frac{1}{k}$. Note that however, the subgradient method cannot be guaranteed to run in polynomial time.

Geometrically, the update of λ in step 6 of Alg. 5.1 as

$$\lambda^{(k+1)} := \lambda^{(k)} + t^{(k)}(x^* - x_{p^{(k)}})$$

can be interpreted as a rotation of the hyperplane induced by ${\lambda^{(k)}}^{\mathrm{T}} x \leq \delta_{\lambda^{(k)}}$ in the direction of the vector $x^* - x_{p^{(k)}}$. An example is illustrated in Fig. 5.3.

5.3 Heuristic for strengthening inequalities with a BDD

In section 5.2 we discussed several approaches to solve the separation problem BDD-SEP. We want to apply the inequalities generated with these methods in a Branch & Cut framework as cuts. However, these approaches naturally tend to generate inequalities which define faces of P_{BDD} with a low dimension. Since we are interested in facets or faces of P_{BDD} with a high dimension, we want to increase their dimension in order to increase the "quality" of the separating hyperplanes. This process is called *strengthening*. Using facet-defining inequalities in Branch & Cut has led to an enormous progress in solving large-scale optimization problems, see e.g. [JRR95]. The standard way to turn a separating hyperplane into a facet-defining inequality (see e.g. [GLSv88]) turned out to be too expensive from a computational point of view. Therefore we developed a heuristic to strengthen inequalities with the help of a BDD, which does not guarantee to produce facets, but can be efficiently implemented.

In the following let $\pi^{\mathrm{T}}x \leq \pi_0$ with $\pi \in \mathbb{R}^d$ be a valid inequality for P_{BDD}. For sake of completeness, we again mention that the right hand side of any inequality, which is valid for P_{BDD}, can be strengthened to $\pi_0 = \max\{\pi^{\mathrm{T}}x \mid x \in P_{\mathrm{BDD}}\}$. The maximum can be computed in linear time in the size of the BDD via optimizing over P_{BDD} (BDD-OPT), see section 4.2. Note that with this method, every inequality can be made tight at at least one vertex of the BDD-polytope.

5.3.1 Increasing the number of tight vertices

We need a method to check if an inequality $\pi^{\mathrm{T}}x \leq \pi_0$ defines a facet of P_{BDD}. W.l.o.g. be the BDD-polytope full-dimensional, and be

$$F := \{x \in P_{\mathrm{BDD}} \mid \pi^{\mathrm{T}}x = \pi_0\}$$

the face of P_{BDD} induced by the given inequality. We define

$$W := \{a \in \mathbb{R}^d \mid \exists \alpha \in \mathbb{R}\ \forall x \in F : a^{\mathrm{T}}x = \alpha\}.$$

W is the vectorspace of all normalvectors, which induce the same face F of P_{BDD}. This leads to the following lemma.

Lemma 5.1. $\pi^{\mathrm{T}}x \leq \pi_0$ *defines a facet of* P_{BDD} *iff* $W =< w >$, *i.e.* $\dim(W) = 1$

In section 4.2.1 we discussed a method, which can also be adapted to work on F instead of P_{BDD}, so we can apply it to compute W and its dimension.

Another way of testing if $\pi^{\mathrm{T}}x \leq \pi_0$ defines a facet of P_{BDD} is to calculate the number of affinely independent $0/1$ vectors in the set F.

Lemma 5.2. $\pi^{\mathrm{T}}x \leq \pi_0$ *defines a facet of* P_{BDD} *iff* $F \cap \{0,1\}^d$ *contains* d *affinely independent vectors, i.e.* $\dim(F) = d - 1$

In section 4.2.2 we gave an algorithm to compute the dimension of a face F, which can be used in this context. Note that in our case $dim(F) = dim(F \cap \{0,1\}^d)$ holds.

For both tests, systems of linear equations have to be solved which is quite expensive in terms of running time. Therefore we choose a different criterion for strengthening a hyperplane. We aim at increasing the number of vertices of P_{BDD} that are tight at $\pi^{\mathrm{T}}x \leq \pi_0$, so chances are high to also increase the dimension of the induced face F.

In the following we try to strengthen a hyperplane induced by the inequality $\pi^{\mathrm{T}}x \leq \pi_0$ along the unit vectors. Remember that every path in the BDD-graph from the root to leaf 1 corresponds to a vertex of P_{BDD}. W.l.o.g. assume that for all $i \in \{1, \ldots, d\}$ the nodes labeled with x_i lie on level i. Given $i \in \{1, \ldots, d\}$ we want to find a new π_i so that the number of longest paths w.r.t. the edge weights given by π increases. We proceed as follows. First we compute the costs of the longest paths which use 0-edges in level i.

$$\alpha_0 := \max_{\substack{x \in P_{\mathrm{BDD}} \\ x_i = 0}} \pi^{\mathrm{T}}x$$

Then we determine the costs of the longest paths using 1-edges in level i.

$$\alpha_1 := \max_{\substack{x \in P_{\mathrm{BDD}} \\ x_i = 1}} \pi^{\mathrm{T}} x$$

If the costs differ, i.e. $\alpha_0 \neq \alpha_1$, we know that the longest paths from root to leaf 1 with x_i freely chosen only use one kind of edge in the level i. In that case, we adjust π_i by adding $\alpha_0 - \alpha_1$ and set $\pi_0 = \alpha_0$. So the longest paths according to the new π now use both kinds of edges in the level i, i.e. the number of longest paths and thus the number of vertices of P_{BDD}, which are tight at the inequality, increased. Note that our heuristic is similar to a strategy known for lifting cover inequalities for the knapsack problem, see e.g. [NW88].

In a Branch & Cut framework it may occur that an inequality separating a given x^* does not separate x^* after strengthening of the coefficient π_i. So after each strengthening step, we check if x^* still violates the inequality, and in case it does not, we undo the strengthening of π_i and choose a different candidate.

Be n the size of the BDD. For the computation of a single π_i, we can use a modified longest path algorithm to compute both costs α_0 and α_1 in one run. Here we again use the fact that we can optimize over P_{BDD} in linear time in the size of the BDD. As we strengthen every coefficient of π, the total running time is $\mathcal{O}(dn)$. If we do not consider permutations of the indices but strengthen the coefficients in the canonical order $1, \ldots, d$ we can use a single call to another modified longest path algorithm, which considers each edge only a constant number of times. So the complexity can be reduced to $\mathcal{O}(n)$.

The strengthened π depends on the order of the indices which we took to strengthen each coefficient π_i. Different permutations of $\{1, \ldots, d\}$ can lead to different strengthened inequalities. So by using different orderings for a given inequality, we can get a family of strengthened inequalities.

5.4 Lifting

Suppose we have generated some cuts with a BDD in a node of the Branch & Cut tree. Recall from section 5.1, that in case we are not in the root node, some variables might be fixed to the values 0 or 1. So we have to consider the decomposition of the index set $\{1, \ldots, d\}$ into three disjoint sets $I_0 \dot\cup I_1 \dot\cup F = \{1, \ldots, d\}$. The fixation of the variables is given by $\forall i \in I_0 : \ x_i = 0$ and $\forall i \in I_1 : \ x_i = 1$. In the following, be $\emptyset \neq F \neq \{1, \ldots, d\}$.

Thus the inequalities, which we generated by solving BDD-SEP, are valid for the face $\mathcal{F}$ of the BDD-polytope, which is defined as

$$\mathcal{F} := P_{\mathrm{BDD}} \cap \bigcap_{\delta \in \{0,1\}} \{x \in [0,1]^d \mid x_i = \delta \ \ \forall i \in I_\delta\}.$$

In this section we show how to extend inequalities which are valid for a face of the BDD-polytope to be valid for the BDD-polytope P_{BDD}. Thereto we adapt a technique called *lifting* (see e.g. [NW88]) to our context.

For a given $i \in I_0$, let $\mathcal{F}_{i \in I_1}$ be the face $\mathcal{F}$ of P_{BDD} constructed by $I_0 \setminus \{i\}$, $I_1 \cup \{i\}$ and be $\mathcal{F}_{i \in F}$ the face constructed by $I_0 \setminus \{i\}$, $F \cup \{i\}$. Then we have the following well-known lemma.

Lemma 5.3. *Be $i \in I_0$, and be the inequality*

$$\sum_{j \in F} \pi_j x_j \leq \pi_0 \tag{5.9}$$

valid for $\mathcal{F}$. Define $\alpha_i := \pi_0 - \zeta$ with ζ given by

$$\zeta := \max_{x \in \mathcal{F}_{i \in I_1} \cap \{0,1\}^d} \sum_{j \in F} \pi_j x_j$$

Then the inequality

$$\alpha_i x_i + \sum_{j \in F} \pi_j x_j \leq \pi_0 \tag{5.10}$$

is valid for $\mathcal{F}_{i \in F}$. In addition, if (5.9) *defines a face of dimension k (resp. a facet) of $\mathcal{F}$, then* (5.10) *defines a face of dimension $k+1$ (resp. a facet) of $\mathcal{F}_{i \in F}$.*

Proof. For $\bar{x} \in \mathcal{F}$ we have

$$\alpha_i \bar{x}_i + \sum_{j \in F} \pi_j \bar{x}_j = \sum_{j \in F} \pi_j \bar{x}_j \leq \pi_0$$

since (5.9) is valid for $\mathcal{F}$. If $\bar{x} \in \mathcal{F}_{i \in I_1}$ then

$$\alpha_i \bar{x}_i + \sum_{j \in F} \pi_j \bar{x}_j = \alpha_i + \sum_{j \in F} \pi_j \bar{x}_j \leq \alpha_i + \zeta = \pi_0$$

holds by the definitions of α_i and ζ.

The inequality (5.9) defines a k-dimensional face of $\mathcal{F}$. So there exist $k+1$ affinely independent points $\bar{x}^l \in \mathcal{F}$ with $l \in \{1, \ldots, k+1\}$, which are tight at (5.9), i.e. they fulfill the inequality with equality. As we have $\bar{x}^l_i = 0$ for each $l \in \{1, \ldots, k+1\}$, each $\bar{x}^l$ is also tight at (5.10). Be $x^* \in \mathcal{F}_{i \in I_1} \cap \{0,1\}^d$ such that $\zeta = \sum_{j \in F} \pi_j x^*_j$. Then, with $\alpha_i = \pi_0 - \zeta$, x^* is tight at (5.10). Since $x^*_i = 1$ and $\bar{x}^l_i = 0$ for each $l \in \{1, \ldots, k+1\}$, these $k+2$ vectors are affinely independent. □

We heavily rely on the fact, that in our case the extreme points of any face of P_{BDD} are binary points, and that we are able to compute the above ζ by optimizing on the BDD in time linear its size (see section 4.2). In practice, we can do this via a modified longest path algorithm on the BDD which finds the corresponding binary $x^* \in \mathcal{F}_{i \in I_1}$.

Consider now a given $i \in I_1$ and define $\mathcal{F}_{i \in I_0}$ as the face $\mathcal{F}$ of P_{BDD} constructed by $I_1 \setminus \{i\}$, $I_0 \cup \{i\}$ and $\mathcal{F}_{i \in F}$ as the face constructed by $I_1 \setminus \{i\}$, $F \cup \{i\}$. Then we have the analogous lemma.

Lemma 5.4. *Be $i \in I_1$, and be the inequality* (5.9) *valid for $\mathcal{F}$. Define $\gamma_i := \zeta - \pi_0$ with ζ given by*

$$\zeta := \max_{x \in \mathcal{F}_{i \in I_0} \cap \{0,1\}^d} \sum_{j \in F} \pi_j x_j.$$

Then the inequality

$$\gamma_i x_i + \sum_{j \in F} \pi_j x_j \leq \gamma_i + \pi_0 \tag{5.11}$$

is a valid for $\mathcal{F}_{i \in F}$. In addition, if (5.9) *defines a face of dimension k (resp. a facet) of $\mathcal{F}$, then* (5.11) *defines a face of dimension $k+1$ (resp. a facet) of $\mathcal{F}_{i \in F}$.*

Proof. The proof is analogous to the one for lemma 5.3. □

Given an inequality which is valid for a face of P_{BDD}, we can apply the above lemmas sequentially to obtain an inequality which is valid for P_{BDD}. The new coefficients α_i and γ_i depend on the order, which is chosen for lifting the variables. By considering different orderings of the variable indices in $I_0 \cup I_1$, it is possible to gain a family of valid inequalities for P_{BDD}.

5.5 Computational results

We investigated the practical strength of the theory which we developed within this work by doing computational experiments. Thereto we implemented all of the methods mentioned in this chapter and integrated them in the separation step of a Branch & Cut algorithm.

Given a 0/1 integer program of the form

$$\begin{array}{rl} \max & c^{\mathrm{T}} x \\ \text{s.t.} & Ax \leq b \\ & x \in \{0,1\}^d \end{array}$$

with $A \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^d$, which we want to solve using Branch & Cut. Assume that we have reached a node of the Branch & Cut tree, and that the point $x^* \in \mathbb{R}^d$ is given as the solution of the current LP relaxation, but is infeasible for the IP. Then we call our separation procedure using BDDs, which works as follows.

(1) **Fix** the variables according to the branching decisions.

(2) **Build the BDD** from some of the constraints of the IP, obeying the fixations of the variables.

(3) **Learn** logical cuts from the BDD structure, if possible, and return them.

(4) **Solve the separation problem** for x^* with the subgradient method.

(5) **Strengthen** the cuts with our heuristic.

(6) **Lift** the strengthened cuts into the original full space and return them.

The cuts that we gain by using BDDs can be used for any 0/1 integer program, even for those, where nothing is known about their structure. We report on our results with MAX-ONES problems and randomly generated 0/1 IPs. They show that we could achieve a considerable speedup on small but hard 0/1 IPs.

5.5.1 Implementation

To evaluate the effectiveness, we implemented our methods in C++. We embedded our separation routine in the cutcallback function of the `CPLEX`[1] `9.0` Branch & Cut framework.

Before we build a BDD for the first time, we use our WOG heuristic from section 3.3.1 on the matrix given in the 0/1 IP for finding a good initial variable order. In terms of finding a variable order which decreases the size of the BDDs, WOG turned out to be slightly better than BOG.

Now that we have chosen an initial variable order for the BDDs, our separation routine can be called in a node of the Branch & Cut tree (see section 5.1). We fix the variables according to the fixation in the branching that led us to this node, which simplifies building the BDD. In addition to that we restrict the constraints that will be used for building the BDD to some of those of the 0/1 IP that are tight at the LP solution of the current node. It is desirable to build the BDD, if possible, at least for all constraints that form the basis of the current LP solution. To identify the "interesting" constraints such that the BDD can be build quickly and still x^* can be cut off, is a task which has to be adjusted for every problem class independently.

For building BDDs we used the `CUDD`[2] `2.4.1` library. In our implementation we build the ROBDD for every constraint of the matrix once with the classical algorithm from section 3.1.1. Then we save these threshold BDDs in a set. According to the constraints chosen for building the BDD in a node of the Branch & Cut tree, we then use the sequential *and*-operator from section 3.2.1 on the according threshold BDDs from the set to build the desired BDD. If the number of nodes exceeds a given limit while building the BDD, we turn on sifting (see section 3.3.2) occasionally. 60.000 proved to be a good node limit for sifting. If the size of the BDD gets too large, which means in our case, more than 1 million nodes, we stop building it. Note that at any step of the separation routine, we have the freedom to return to the calling node of the Branch & Cut tree without providing a separating cut, although this is not desired. In that case the Branch & Cut framework will not resolve the current node but branch on it.

Next we check, if we can derive logical cuts which we have presented in section 5.1.1. If the BDD is empty, we return the exclusion cut. In case, there are variables in the graph representation of the BDD whose nodes only have outgoing edges of one kind, we return the implication cuts. In practice however it showed, that these kinds of cuts are seldom generated.

As we heavily rely on fast optimization over the BDD-polytope (see section 4.2), we implemented an efficient version of an acyclic shortest path algorithm on the BDD-datastructure used in `CUDD`.

We implemented all approaches which we developed for solving the separation problem BDD-SEP, i.e. the separation LP (5.4) from section 5.2.2, the cutting plane approach with the LP (5.8) from section 5.2.3, and the subgradient method for the Lagrangean relaxation as given in Alg. 5.1 in section 5.2.4. Although the subgradient method cannot be guaranteed to run in polynomial time, we observed that it outperforms linear programming

[1]`http://www.ilog.com/products/cplex`, CPLEX homepage, ILOG

[2]`http://vlsi.colorado.edu/~fabio/CUDD`, CU Decision Diagram package homepage, F. Somenzi, 2005

methods for BDD-SEP by far. This is why we rely in our code on a variant of this method. Due to numerical problems the subgradient method sometimes does not terminate. We investigated the step length $t^{(k)}$ and found out, that increasing the denominator by 1 in every s-th iteration leads to a higher numerical stability, where $s = 5$ showed to be a good value for most of the cases. If we cannot find a separating hyperplane after 2000 iterations we stop.

In order not to risk numerical stability of the LP solving process by introducing cuts with arbitrary coefficients, we first make our cuts integer. Thereto we multiply them with an adequate integer value and round them. After that we further strengthen the right hand side and the coefficients as described in section 5.3. In almost all of the cases the resulting integer hyperplanes are still separating the current LP solution x^* from the 0/1 IP.

If some variables were fixed while building the BDD, the generated cuts are only valid for the face of P_{BDD}, which corresponds to the given fixations. To make these cuts valid for P_{BDD} we sequentially lift them with the procedures given in section 5.4.

5.5.2 Benchmark sets

MIPLIB

From the MIPLIB 3.0 [BBI92] we consider those problems where all variables are 0/1.

MAX-ONES

Satisfiability problems notoriously produce hard to solve 0/1 IPs [ACF07]. Therefore we investigated SAT instances and converted them to MAX-ONES problems. A given SAT-instance over d boolean variables and a set of clauses $C_1, \ldots, C_k$ can easily be transformed into a 0/1 IP representing a MAX-ONES problem by converting each clause to a linear constraint of the form $\sum_i x_i + \sum_j (1 - x_j) \geq 1$ and adding the objective function $\max \sum_{i=1}^{d} x_i$. From a SAT competition held in 1992 [BB93], we took the hfo instances. The 5cnf instances are competition benchmarks of SAT-02, and the remaining SAT instances are competition benchmarks from SAT-03. These instances can be found in the SATLIB [HS00].

Randomly generated 0/1 IPs

Additionally we are interested in how our code performs on problems with less or without any structure. Therefore we randomly generated 0/1 IPs the following way: an entry in the matrix A, the right hand side b and the objective function c gets a nonzero value with probability p. This value is randomly chosen from the integers with absolute value less or equal $c_{\max}$.

5.5.3 Results

Our experiments have been performed on a Linux system with kernel 2.6 on an Xeon CPU with 3.06 GHz and 4 GB memory. Every investigated problem was solved to optimality or proven to be infeasible.

We compare the following implementations for solving 0/1 integer programs. On the one hand we run `CPLEX`[3] `9.0` with the default values (`cplex`), i.e. it did presolving and used all types of built-in separation cuts, which are clique cuts, cover cuts, disjunctive cuts, flow cover cuts, flow path cuts, Gomory fractional cuts, GUB cuts, implied bound cuts, and mixed integer rounding cuts. On the other hand we integrated our separation routine with BDDs in `CPLEX`'s Branch & Cut framework (`bcBDD`), but switched off presolve and all built-in cuts. The reason for switching off presolve is, that we sometimes encountered problems working on the presolved model. We also tried to switch off presolve for the benchmarks made with the default `cplex`. It showed, that presolving the randomly generated IPs does not really influence the running times but switching off presolve for the MAX-ONES instances increased `cplex`'s running times.

For the instances from the *MIPLIB* we examined the influence of our cuts on the size of the Branch & Cut tree. Therefore we additionally took a pure Branch & Bound approach (`bb`) into account, which we realized by using `CPLEX` with default values and all built-in cuts switched off. Table 5.1 gives the results. In the first part of the table problems are shown where the size of the Branch & Cut tree with the BDD cuts is smaller than with the cuts built into `CPLEX`. In the second part the BDD cuts reduce the number of nodes in comparison to the Branch & Bound tree without any cuts. The third part consists of problems where no reduction of the tree size could be achieved.

Regarding the runtimes of `bcBDD` on the MIPLIB instances we cannot compare to `cplex`, since most of the problems have a large number of variables and most of the time is spent building the BDDs. Therefore we considered to make this comparison on small and hard 0/1 integer programs. We chose MAX-ONES problems and randomly generated 0/1 IPs. In the corresponding tables the runtimes are the total user times given in seconds. We computed the speedup as 1 minus the ratio of `bcBDD`'s runtime divided by `cplex`'s runtime. The corresponding figures show the instances sorted in ascending order of `cplex`'s runtime.

For the *MAX-ONES* instances we found out, that generating nearly all of our cuts in the root node is the most promising strategy. Using too few constraints to build the BDD resulted in weaker cutting planes. In practice it showed that 70% of the constraints, that are tight at the current LP solution, is a good threshold for generating cuts with an adequate quality while building the BDD does not consume too much time. The table 5.2 and the figure 5.4 show the runtimes for the instances of SAT-02/SAT-03. For the 5cnf instances, the average speedup is 26.09%, whereas for the instances from SAT-03, the average speedup is only 0.85%. For these instances it is hard to find cuts to prove infeasibility. The runtimes for the hfo instances are presented in table 5.3 and figure 5.5. For 145 of the 160 hfo instances we obtain faster running times than `cplex`. The average of the overall speedup for the hfo instances is 18.31% with a standard deviation of 14.44%.

For *randomly generated 0/1 IPs* building the BDDs is harder as the constraints have no structure. We generated our cuts deeper in the Branch & Cut tree and lifted them afterwards. Furthermore we only used 20% of the constraints that form the basis of the LP solution in the current Branch & Cut node. Table 5.4 and figure 5.6 show the results. For the randomly generated 0/1 IPs we achieved an average speedup of 34.23%.

[3] `http://www.ilog.com/products/cplex`, CPLEX homepage, ILOG

Name	Var.	Cons.	bb	cplex	bcBDD
stein9	9	13	6	8	1
stein15	15	36	77	83	1
stein27	27	118	3789	3844	3009
bm23	27	20	114	105	63
p0040	40	23	1	1	1
stein45	45	331	49556	57851	54234
enigma	100	21	424	4335	30
air01	771	23	2	1	1
lp4l	1086	85	48	17	7
l152lav	1989	97	617	617	167
mod010	2655	146	21	23	16
p6000	6000	2176	10	100	2
cap6000	6000	2176	10	100	2
p0033	33	16	100	7	44
pipex	48	25	691	31	462
p0282	282	241	102	58	89
p0291	291	252	20	1	17
mod008	319	6	2355	380	2064
air02	6774	50	42	1	2
air06	8627	825	6	2	4
mitre	10724	2054	58	1	50
sentoy	60	30	104	96	116
lseu	89	28	3210	140	4240
p0201	201	133	297	172	592
p0548	548	176	2049	7	2059
air05	7195	426	672	523	820
air04	8904	823	1147	249	1366
air03	10757	124	2	1	2

Table 5.1: Comparison of the number of nodes of the Branch & Cut trees. Column `bb` gives the number of nodes of a pure Branch & Bound approach. In the first part the size of the Branch & Cut tree with the BDD cuts is smaller than with the cuts built into `cplex`. In the second part the BDD cuts reduce the number of nodes in comparison to the Branch & Bound tree without any cuts. The third part consists of problems where `bcBDD` could not achieve a reduction of the tree size.

Name	Var.	Cons.	satisfiable	cplex	bcBDD	Speedup
5cnf_3800_50f1	50	760	yes	55.59	35.21	36.66 %
5cnf_3900_060	60	936	no	5000.01	3519.79	29.60 %
5cnf_3900_070	70	1092	yes	4523.13	3524.89	22.07 %
5cnf_4000_50f1	50	800	no	183.55	180.21	1.82 %
5cnf_4000_50f7	50	800	no	252.49	240.19	4.87 %
5cnf_4000_50t1	50	800	yes	24.21	12.81	47.09 %
5cnf_4000_50t3	50	800	yes	106.74	89.66	16.00 %
5cnf_4000_50t8	50	800	yes	125.63	109.32	12.98 %
5cnf_4000_60t5	60	960	yes	3905.54	3458.66	11.44 %
5cnf_4100_50f1	50	820	no	291.00	206.07	29.19 %
5cnf_4100_50f2	50	820	no	237.47	171.19	27.91 %
5cnf_4100_50f3	50	820	no	253.30	153.63	39.35 %
5cnf_4100_50f5	50	820	no	259.19	166.43	35.79 %
5cnf_4100_50f7	50	820	no	380.19	257.91	32.16 %
5cnf_4100_50t1	50	820	no	242.31	134.71	44.41 %
icosahedron	30	192	no	184.35	186.81	-1.39 %
marg2x5	35	120	no	22.52	23.51	-4.40 %
marg2x6	42	144	no	207.22	237.38	-14.55 %
marg2x7	49	168	no	3371.32	3330.37	1.21 %
marg3x3add4	37	160	no	453.39	414.66	8.54 %
urqh1c2x4	35	216	no	492.38	464.90	5.58 %
urqh2x3	31	240	no	465.25	413.98	11.02 %

Table 5.2: Results for the SAT-02/SAT-03 instances. The runtimes are given in seconds. The Speedup is computed as 1 minus bcBDD's runtime divided by cplex's runtime.

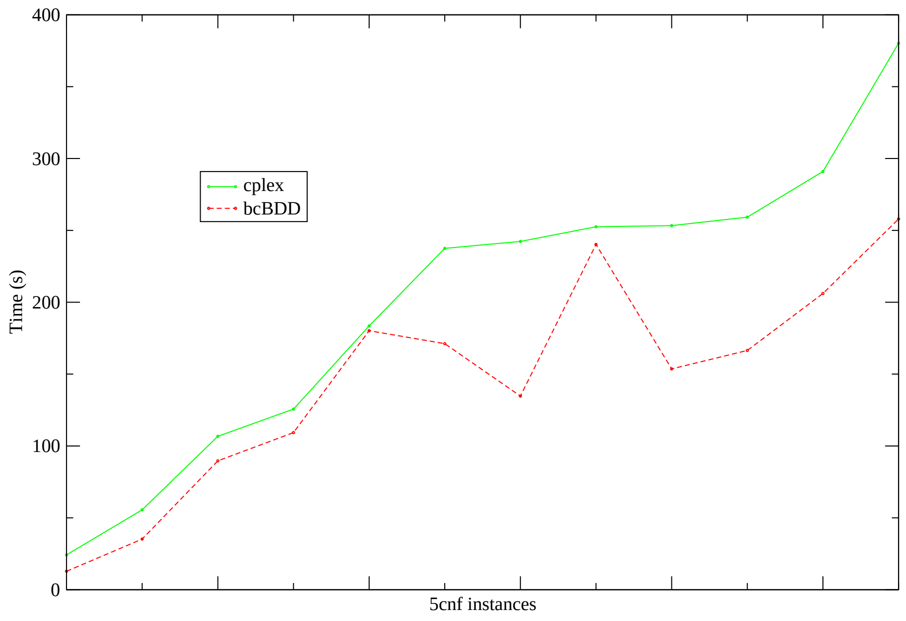


Figure 5.4: Results for the 5cnf instances. The instances are sorted in ascending order of cplex's runtime.

Name	Var.	Cons.	satisfiable	`cplex`	`bcBDD`	Speedup
hfo5	55	1163	no	3615.93	2510.11	27.32 %
				(770.64)	*(437.74)*	*(21.36 %)*
hfo5	55	1163	yes	1917.11	1399.99	22.88 %
				(1108.12)	*(764.53)*	*(17.60 %)*
hfo6	40	1745	no	966.84	771.48	19.97 %
				(77.72)	*(58.14)*	*(5.94 %)*
hfo6	40	1745	yes	529.44	417.65	21.71 %
				(256.00)	*(222.52)*	*(19.69 %)*
hfo7	32	2807	no	662.65	557.29	15.33 %
				(60.98)	*(23.68)*	*(7.47 %)*
hfo7	32	2807	yes	346.32	302.31	8.93 %
				(193.71)	*(156.45)*	*(10.37 %)*
hfo8	27	4831	no	690.39	592.29	14.02 %
				(36.73)	*(19.20)*	*(4.66 %)*
hfo8	27	4831	yes	352.31	297.77	16.29 %
				(211.31)	*(171.71)*	*(11.10 %)*

Table 5.3: Results for the hfo instances. The runtimes are given in seconds and show average values taken over 20 different instances of each type. The standard deviation is given in brackets. The Speedup is computed as 1 minus `bcBDD`'s runtime divided by `cplex`'s runtime.

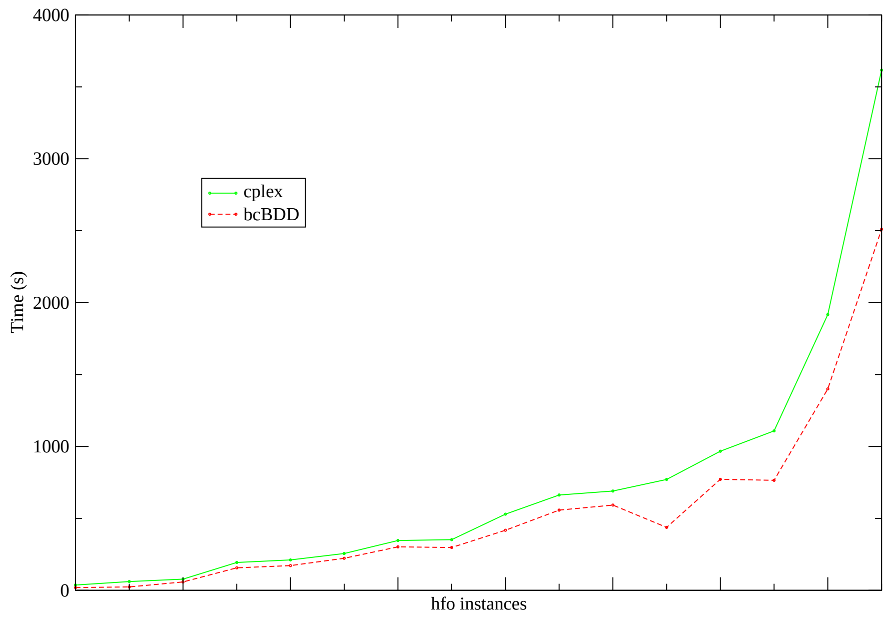


Figure 5.5: Results for the hfo instances. The instances are sorted in ascending order of `cplex`'s runtime.

Name	Var.	Cons.	satisfiable	p	$c_{\max}$	`cplex`	`bcBDD`	Speedup
rand50_00	50	40	no	0.6	15	28.30	17.05	39.75 %
rand50_01	50	50	yes	0.6	13	49.17	17.17	65.08 %
rand50_02	55	50	no	0.6	15	41.53	33.43	19.50 %
rand55_00	55	55	no	0.7	17	137.50	108.69	20.95 %
rand55_01	55	55	yes	0.7	17	85.20	78.61	7.73 %
rand60_00	60	60	no	0.6	13	151.65	85.60	43.55 %
rand60_01	60	60	no	0.6	13	104.58	92.49	11.56 %
rand60_02	60	60	no	0.6	13	237.59	173.62	26.92 %
rand60_03	60	60	no	0.6	13	191.10	134.90	29.41 %
rand60_04	60	60	no	0.6	13	155.90	106.82	31.48 %
rand60_05	60	60	no	0.6	13	285.83	155.83	45.48 %
rand60_06	60	60	no	0.6	13	678.75	406.58	40.10 %
rand60_07	60	60	yes	0.6	13	84.33	56.26	33.29 %
rand60_08	60	60	no	0.6	13	79.10	78.04	1.34 %
rand70_00	70	70	no	0.6	12	511.62	280.85	45.11 %
rand80_00	80	80	yes	0.6	4	89.47	31.30	65.02 %
rand90_00	90	90	yes	0.4	4	192.36	85.45	55.58 %

Table 5.4: Results for the random 0/1 IP instances. These were generated the following way: an entry in the matrix A, the right hand side b and the objective function c gets a nonzero value with probability p, where the value is randomly chosen from the integers with absolute value less or equal $c_{\max}$. The runtimes are given in seconds. The Speedup is computed as 1 minus `bcBDD`'s runtime divided by `cplex`'s runtime.

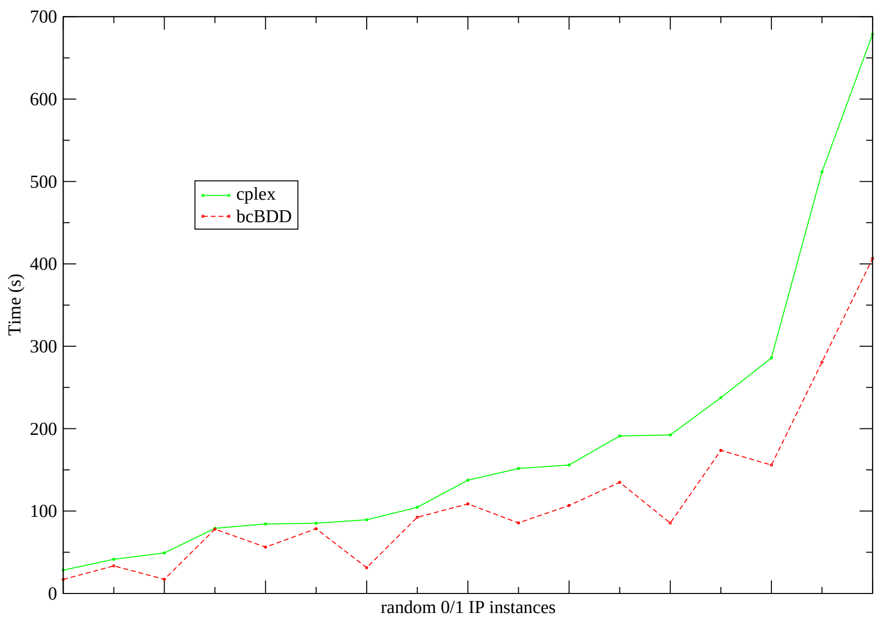


Figure 5.6: Results for the random 0/1 IP instances. The instances are sorted in ascending order of `cplex`'s runtime.

Summary

In this thesis we develop methods for the integration of Binary Decision Diagrams (BDDs for short) in 0/1 integer programming and related polyhedral investigations. BDDs are a datastructure represented by a directed acyclic graph, which aims at a compact and efficient representation of boolean functions. Since their significant extension in 1986 in the famous paper by Bryant [Bry86] they have received a lot of attention in fields like computational logics and hardware verification. They are used as an industrial strength tool, e.g. in VLSI design [MT98].

One class of BDDs are the so-called threshold BDDs. A threshold BDD represents in a compact way the set of 0/1 vectors which are feasible for a given linear constraint. As there is an obvious relation to the Knapsack problem, and thus to 0/1 integer programming in general, we study this class of BDDs. The classical algorithm for building a threshold BDD (see e.g. [Weg00]) is in principle similar to dynamic programming for solving a Knapsack problem (see e.g. [Sch86]). It is a recursive method, which ensures a unique representation of the output by applying certain rules while building the BDD. In particular, isomorphic subgraphs will be detected after being built and then deleted or merged again. We develop a new algorithm for building a threshold BDD which is output-sensitive. More precisely, our algorithm constructs exactly as many nodes as the final BDD consists of and does not need any extra memory.

For many problems in combinatorial optimization there exists a 0/1 integer programming (0/1 IP) formulation, i.e. a set of linear constraints together with a linear objective function and a restriction of the variables to 0 or 1. The natural way for building a BDD for such a problem is the following. First build a threshold BDD for each constraint separately, and then use a pairwise *and*-operator on the set of BDDs in a sequential fashion, until one BDD is left. This way, intermediate BDDs will be constructed which can have a representation size, that is several times larger than that of the final BDD. We give an *and*-operation that synthesizes all threshold BDDs in parallel, which is a novelty. Thereby we overcome the problem of explosion in size during computation.

In addition, we look at the connection between BDDs and 0/1 integer programming from the opposite point of view, i.e. how 0/1 integer programming can be applied to the field of threshold BDDs. We develop for the first time a 0/1 IP, whose optimal solution gives the size and the optimal variable order of a threshold BDD. Usually, the variable ordering spectrum of a BDD is not computable. With the help of this 0/1 IP, we are now able to compute the variable ordering spectrum of a threshold BDD.

In general, 0/1 integer programming problems are hard to solve although they might

have a small representation size. The transformation of such a problem to a BDD shifts these properties, i.e. the representation size possibly gets large while the optimization problem becomes fairly easy to solve. In fact, it reduces to a shortest path problem on a directed acyclic graph which can be solved in linear time in the number of nodes of the graph. This is the main motivation for the investigation of using BDDs in 0/1 integer programming. Apart from optimization, a lot of other tasks can be efficiently tackled if the BDD for a 0/1 integer programming problem could be build.
In polyhedral studies of 0/1 polytopes two prominent problems exist.

One is the vertex enumeration problem: Given a system of inequalities, count or enumerate its feasible 0/1 points. In addition, if a linear objective function is given, compute one optimal solution, or count or enumerate all optimal solutions. We developed a freely available tool called `azove`, which is capable of vertex counting and enumeration for 0/1 polytopes and can easily be extended to optimization. It is based on our techniques for building BDDs. Computational results show that our tool is currently the fastest available. On some instances, it is several orders of magnitude faster than existing codes.

Another one is the convex hull problem: Given a set of 0/1 points in dimension d, enumerate the facets of the corresponding polytope. We extend the gift-wrapping algorithm with BDDs to solve the facet enumeration problem. In this context BDDs are used to rotate a facet-defining inequality along a ridge to find a new facet. As shown by computational results, our approach can be recommended for 0/1 polytopes whose facets contain few vertices.

Branch & Cut is an effective method for solving 0/1 IPs. In theory it is also possible to solve such problems by building the according BDD. But the disadvantage of BDDs is, that building the entire BDD is in practice hard. The running time of Branch & Cut depends on many things, among which are the "quality" of separated cutting planes. A further point of interest is the generation of cutting planes from not only one constraint of the problem formulation but from two or an arbitrary set of constraints. We combine advantages of both fields to develop a fast Branch & Cut algorithm for 0/1 IPs. For the first time we apply BDDs for separation in a Branch & Cut framework and develop further methods, which are necessary for full integration. The computational results which we achieved on MAX-ONES instances and randomly generated 0/1 IPs show, that we developed code which is competitive with state-of-the-art MIP solvers.

For all of the above mentioned problems from the various fields, we provide efficient algorithms and implementations based on BDDs. This stresses the practical point of view of this thesis. Our work shows that BDDs can serve as a powerful tool for 0/1 integer programming and related polyhedral investigations.

Zusammenfassung

In dieser Arbeit entwickeln wir Methoden zur Integration von Binary Decision Diagrams (im Folgenden kurz BDDs genannt) in 0/1 ganzzahlige Programmierung und zur Untersuchung von dazugehörigen Polytopen. BDDs sind eine Datenstruktur, die eine boolsche Funktion in einem speziellen gerichteten azyklischen Graphen kompakt und effizient repräsentiert. Seit ihrer signifikanten Erweiterung im Jahr 1986 in der bedeutenden Veröffentlichung von Bryant [Bry86] haben sie enorme Beachtung in Bereichen wie Computational Logic und Hardware Verifikation erhalten. Sie werden als Hilfsmittel im industriellen Rahmen, z.B. im Platinen (VLSI) Design [MT98], eingesetzt.

Eine Klasse von BDDs sind die so genannten Threshold BDDs. Ein Threshold BDD stellt die Menge von 0/1 Vektoren kompakt dar, die zulässige Lösungen einer gegebenen linearen Ungleichung sind. Somit besteht offensichtlich eine Verbindung zum Knapsack Problem, und damit zur 0/1 ganzzahligen Programmierung im Allgemeinen, weswegen wir diese Klasse von BDDs untersuchen. Der klassische Algorithmus zum Bauen eines Threshold BDDs (siehe z.B. [Weg00]) ist im Prinzip ähnlich zur Dynamischen Programmierung für das Knapsack Problem (siehe z.B. [Sch86]). Es handelt sich hierbei um eine rekursive Methode, die eine eindeutige Darstellung der Ausgabe dadurch sicherstellt, dass bestimmte Regeln während des Bauens des BDDs angewendet werden. Im Speziellen werden isomorphe Untergraphen nach dem Bauen erkannt und anschließend gelöscht oder zusammengeführt. Wir entwickeln einen neuen Algorithmus zum Bauen eines Threshold BDDs, der output-sensitiv ist. Genauer gesagt baut unser Algorithmus genau so viele Knoten, wie der resultierende BDD benötigt, und braucht darüberhinaus keinen zusätzlichen Speicher.

Für einige Probleme aus der Kombinatorischen Optimierung gibt es eine 0/1 ganzzahlige Formulierung, d.h. sie können durch eine Menge von linearen Ungleichungen zusammen mit einer linearen Zielfunktion und der Einschränkung der Variablen auf 0 oder 1 beschrieben werden. Natürlicherweise baut man den BDD für solche Probleme wie folgt auf. Zuerst baut man den Threshold BDD für jede Ungleichung separat. Anschließend verwendet man sequentiell einen *und*-Operator paarweise auf der Menge der BDDs, bis nur noch ein BDD übrig bleibt. Auf diese Weise werden BDDs als Zwischenresultate gebaut, die eine Darstellungsgröße haben können, die um ein Vielfaches größer ist als die des resultierenden BDDs. Wir beschreiben einen *und*-Operator, der alle Threshold BDDs parallel zusammenführt, was eine Neuheit darstellt. Dadurch verhindern wir Probleme mit der Explosion der Darstellungsgröße.

Zusätzlich betrachten wir die Verbindung zwischen BDDs und 0/1 ganzzahliger Programmierung vom entgegengesetzten Standpunkt aus, d.h., wir schauen uns an, wie man

0/1 ganzzahlige Programmierung für Threshold BDDs einsetzen kann. Wir entwickeln zum ersten Mal ein 0/1 ganzzahliges Programm, dessen optimale Lösung die Größe und die optimale Variablenordnung eines Threshold BDDs angibt. Für gewöhnlich existiert kein Verfahren zum Berechnen des Variablenordnung-Spektrums eines BDDs. Mit Hilfe unseres 0/1 ganzzahligen Programms sind wir nun in der Lage, das Variablenordnung-Spektrum eines Threshold BDDs zu berechnen.

Im Allgemeinen sind Probleme der 0/1 ganzzahligen Programmierung schwierig zu lösen, obwohl sie meistens eine kleine Darstellungsgröße haben. Die Umwandlung eines solchen Problems in einen BDD verschiebt diese Eigenschaften, d.h., die Darstellungsgröße vergrößert sich wahrscheinlich, wobei das Optimierungsproblem extrem leicht zu lösen wird. Es lässt sich auf ein Kürzeste-Wege Problem in einem gerichteten azyklischen Graphen reduzieren, welches in linearer Zeit in der Anzahl der Knoten des Graphen gelöst werden kann. Aus diesem Punkt heraus entsteht die Motivation, die Benutzung von BDDs in der 0/1 ganzzahligen Programmierung zu untersuchen. Über die Optimierung hinaus können viele weitere Aufgabenstellungen effizient angegangen werden, wenn man den BDD für ein 0/1 ganzzahliges Programmierungsproblem aufbauen konnte.
Im Bereich der polyedrischen Untersuchungen von 0/1 Polytopen gibt es zwei bekannte Probleme.

Eines ist das Knoten-Enumerierung-Problem: Gegeben ein System von Ungleichungen, zähle oder enumeriere alle gültigen 0/1 Punkte. Falls zusätzlich eine lineare Zielfunktion gegeben ist, berechne eine optimale Lösung oder zähle oder enumeriere alle optimalen Lösungen. Wir haben ein frei verfügbares Programm namens `azove` entwickelt, dass für 0/1 Polytope Knoten zählt und enumeriert sowie leicht um Optimierung erweitert werden kann. Es basiert auf den von uns entwickelten Techniken zum Bauen von BDDs. Rechenresultate zeigen, dass unser Programm momentan das schnellste zur Verfügung stehende ist. Auf manchen Instanzen ist es um einige Größenordnungen schneller als bestehende Programme.

Ein weiteres Problem ist das Konvexe-Hülle-Problem: Gegeben eine Menge von 0/1 Punkten in Dimension d, enumeriere die Facetten des dazugehörigen Polytops. Wir haben den Gift-Wrapping Algorithmus mit BDDs erweitert, um dieses Problem zu lösen. Hier werden BDDs eingesetzt, um Facetten-definierende Ungleichungen entlang einer Kante zu drehen, um eine neue Facette zu finden. Mit Hilfe von Rechenresultate konnten wir zeigen, dass unser Ansatz gut für 0/1 Polytope geeignet ist, deren Facetten wenige Knoten enthalten.

Branch & Cut ist eine effiziente Methode zum Lösen von 0/1 ganzzahligen Programmen. Theoretisch ist es auch möglich, solche Probleme durch das Bauen des entsprechenden BDDs zu lösen. Der Nachteil von BDDs ist allerdings, dass das Bauen des vollständigen BDDs in der Praxis schwierig ist. Die Laufzeit von Branch & Cut hängt von vielen Dingen wie z.B. der "Qualität" von separierten Schnittebenen ab. Ein weiterer interessanter Punkt ist die Generierung von Schnittebenen aus nicht nur einer Ungleichung der Problembeschreibung, sondern aus zwei oder einer beliebigen Menge von Ungleichungen. Wir kombinieren die Vorteile beider Ansätze und entwickeln einen schnellen Branch & Cut Algorithmus für 0/1 ganzzahlige Probleme. Zum ersten Mal verwenden wir BDDs zur Separierung in einem Branch & Cut Framework und entwickeln alle Methoden, die zur vollständigen Integration nötig sind. Die Rechenresultate, die wir auf MAX-ONES Instanzen und zufällig generierten 0/1 ganzzahligen Programmen erreichen, zeigen, dass das von uns entwickelte Programm vergleichbar mit modernen Lösern für gemischt ganzzahlige Programme ist.

Für alle oben genannten Probleme der unterschiedlichen Bereiche haben wir effiziente Algorithmen und Implementationen basierend auf BDDs entwickelt. Dies betont den praktischen Blickwinkel dieser Thesis. Unsere Arbeit zeigt, dass BDDs als ein starkes Hilfsmittel für die 0/1 ganzzahlige Programmierung und die Untersuchung von dazugehörigen Polytopen dienen können.

Bibliography

[ACF07] G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch & cut framework: a computational study with $\{0, 1/2\}$-cuts. Preprint. To appear in: INFORMS Journal on Computing, 19(2), 2007.

[AF92] D. Avis and K. Fukuda. A pivoting algorithm for convex hull and vertex enumeration of arrangements and polyhedra. *Discrete & Computational Geometry*, 8(3):295–313, 1992.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[ALWW07] K. Andersen, Q. Louveaux, R. Weismantel, and L. A. Wolsey. Inequalities from two rows of a simplex tableau. In M. Fischetti and D. P. Williamson, editors, *Proceedings of the 12th International Conference on Integer Programming and Combinatorial Optimization (IPCO'07)*, volume 4513 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows.* Prentice Hall Inc., 1993.

[Avi00] D. Avis. lrs: A revised implementation of the reverse search vertex enumeration algorithm. In G. Kalai and G. M. Ziegler, editors, *Polytopes – Combinatorics and Computation*, pages 177–198. Birkhäuser, 2000.

[Bal75] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[Bal01] E. Balas. Projection and lifting in combinatorial optimization. In M. Jünger and D. Naddef, editors, *Computational combinatorial optimization*, volume 2241 of *Lecture Notes in Computer Science*, pages 26–56. Springer, 2001.

[Bar94] A. Barvinok. Polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19:769–779, 1994.

[BB93] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.

[BBEW05] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. BDDs in a branch and cut framework. In S. Nikoletseas, editor, *Experimental and Efficient Algorithms, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 452–463, Santorini, Greece, May 2005. Springer.

[BBI92] R. E. Bixby, E. A. Boyd, and R. R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25(2), 1992.

[BCC93] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[BE07] M. Behle and F. Eisenbrand. 0/1 vertex and facet enumeration with BDDs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proceedings of the ninth Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 158–165, New Orleans, USA, January 2007. SIAM.

[Beh07a] M. Behle. On threshold BDDs and the optimal variable ordering problem. In A. Dress, Y. Xu, and B. Zhu, editors, *Proceedings of the first international Conference on Combinatorial Optimization and Applications (COCOA'07)*, volume 4616 of *Lecture Notes in Computer Science*, pages 124–135, Xi'an, China, August 2007. Springer.

[Beh07b] M. Behle. On threshold BDDs and the optimal variable ordering problem. *Journal of Combinatorial Optimization*, 2007. to appear.

[BFG$^+$99] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and practice - closing the gap. In *Proceedings of the 19th IFIP TC7 Conference on System Modelling and Optimization: Methods, Theory and Applications*, volume 174, pages 19 – 50, 1999.

[BFM98] D. Bremner, K. Fukuda, and A. Marzetta. Primal-dual methods for vertex and facet enumeration. *Discrete & Computational Geometry*, 20(3):333–357, 1998.

[BL98] M. R. Bussieck and M. E. Lübbecke. The vertex set of a 0/1-polytope is strongly P-enumerable. *Computational Geometry*, 11(2):103–109, 1998.

[Bre96] D. Bremner. Incremental convex hull algorithms are not output sensitive. In *Proceedings of the 7th International Symposium on Algorithms and Computation (ISAAC'96)*, volume 1178 of *Lecture Notes in Computer Science*, pages 26–35. Springer, 1996.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bry91] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[CF04] G. Codato and M. Fischetti. Combinatorial benders' cuts. In D. Bienstock and G. Nemhauser, editors, *Proceedings of the 10th International Conference on Integer Programming and Combinatorial Optimization (IPCO'04)*, volume 3064 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2004.

[Cha93] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete and Computational Geometry*, 10:377–409, 1993.

[CJP83] H. Crowder, E. J. Johnson, and M. Padberg. Solving large-scale 0-1 linear programming problems. *Operations Research*, 31(5):803–834, 1983.

[CK70] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the Association for Computing Machinery*, 17(78):78–86, 1970.

[CPS90] T. M. Cavalier, P. M. Pardalos, and A. L. Soyster. Modeling and integer programming techniques applied to propositional calculus. *Computers and Operations Research*, 17(6):561–570, 1990.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4:387–421, 1989.

[DDG98] R. Drechsler, N. Drechsler, and W. Günther. Fast exact minimization of BDDs. In *Proceedings of the 35th Design Automation Conference (DAC'98)*, pages 200–205, 1998.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2:393–410, 1954.

[Ebe03] R. Ebendt. Reducing the number of variable movements in exact BDD minimization. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'03)*, volume 5, pages 605–608, 2003.

[EGD03] R. Ebendt, W. Günther, and R. Drechsler. An improved branch and bound algorithm for exact BDD minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(12):1657–1663, 2003.

[EGD04] R. Ebendt, W. Günther, and R. Drechsler. Combining ordered best-first search with branch and bound for exact BDD minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'04)*, pages 875–878, 2004.

[FOH93] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 38–41, 1993.

[FP96] K. Fukuda and A. Prodon. Double description method revisited. In *Selected papers from the 8th Franco-Japanese and 4th Franco-Chinese Conference on Combinatorics and Computer Science (CCS'95)*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer, 1996.

[FS90] S. Friedman and K. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, 1990.

[GJ00] E. Gawrilow and M. Joswig. Polymake: A framework for analyzing convex polytopes. In G. Kalai and G. M. Ziegler, editors, *Polytopes – Combinatorics and Computation*, pages 43–74. Birkhäuser, 2000.

[GK06] R. Gillmann and V. Kaibel. Revlex-initial 0/1-polytopes. *Journal of Combinatorial Theory*, Series A(113):799–821, 2006.

[GLS81] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[GLSv88] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics.* Springer, 1988.

[Gom58] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[HJP75] P. L. Hammer, E. Johnson, and U. N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[HKB04] M. Herbstritt, T. Kmieciak, and B. Becker. On the impact of structural circuit partitioning on SAT-based combinational circuit verification. In *Proceedings of 5th IEEE International Workshop on Microprocessor Test and Verification (MTV'04)*, 2004.

[Hoo04] J. N. Hooker. Planning and scheduling by logic-based benders decomposition. Preprint. To appear in: Operations Research, 2004.

[HS00] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent and T. Walsh, editors, *Satisfiability in the year 2000*, pages 283–292. IOS Press, 2000.

[HTKY97] K. Hosaka, Y. Takenaga, T. Kaneda, and S. Yajima. Size of ordered binary decision diagrams representing threshold functions. *Theoretical Computer Science*, 180:47–60, 1997.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD'91)*, pages 472–475, 1991.

[JKS93] S.-W. Jeong, T.-S. Kim, and F. Somenzi. An efficient method for optimal BDD ordering computation. In *Proceedings of the International Conference on VLSI and CAD (ICVC'93)*, pages 252–256, 1993.

[JRR95] M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In *Handbook on Operations Research and Management Science*, volume 7, pages 225–330. Elsevier, 1995.

[KP80] R. M. Karp and C. H. Papadimitriou. On linear characterizations of combinatorial optimization problems. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS'80)*, pages 1–9. IEEE, 1980.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell Systems Technical Journal*, 38:985–999, 1959.

[LHTY04] J. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4):1273–1302, 2004.

[LPV93] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. FGILP: an integer linear program solver based on function graphs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 685–689, 1993.

[LPV94] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and functional decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):959–975, 1994.

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games II*, volume 8 of *Annals of Mathematics Studies*, pages 51–73. Princeton University Press, 1953.

[MT98] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design.* Springer, 1998.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley, 1988.

[Pap81] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.

[PR81] M. W. Padberg and M. R. Rao. The russian method for linear programming III: Bounded integer programming. Technical Report 81-39, New York University, Graduate School of Business and Administration, 1981.

[PRW85] M. W. Padberg, T. J. Van Roy, and L. A. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, 33(4):842–861, 1985.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[PW84] M. W. Padberg and L. A. Wolsey. Fractional covers for forests and matchings. *Mathematical Programming*, 29(1):1–14, 1984.

[Rot92] G. Rote. Degenerate convex hulls in high dimensions without extra storage. In *Proceedings of the 8th annual Symposium on Computational Geometry (SoCG'92)*, pages 26–32, 1992.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 42–47, 1993.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley, 1986.

[Sei81] R. Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical Report 81-14, University of British Columbia, 1981.

[Sei97] R. Seidel. Convex hull computations. In J. Goodman and J. O'Rouke, editors, *Handbook of Discrete and Computational Geometry*, chapter 19. CRC Press, 1997.

[SW93] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48:139–144, 1993.

[Swa85] G. F. Swart. Finding the convex hull facet by facet. *Journal of Algorithms*, 6:17–48, 1985.

[Urq87] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[Weg00] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2000.

[Wol75] L. A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

[Zie95] G. M. Ziegler. *Lectures on Polytopes*. Springer, 1995.