Carnegie Mellon
Tepper
SCHOOL OF BUSINESS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
INDUSTRIAL ADMINISTRATION
(ALGORITHMS, COMBINATORICS AND OPTIMIZATION)

Titled

"Decision Diagram Relaxations for Integer Programming"

Presented by

Christian Tjandraatmadja

Accepted by

Willem-Jan van Hoeve	**4/26/18**
Chair: Prof. Willem-Jan van Hoeve	**Date**

Approved by The Dean

Robert M. Dammon	**4/28/18**
Dean Robert M. Dammon	**Date**

Decision Diagram Relaxations for Integer Programming

Christian Tjandraatmadja

April 2018

Tepper School of Business
Carnegie Mellon University

Submitted to the Tepper School of Business
in Partial Fulfillment of the Requirements for the Degree of
Doctor in Algorithms, Combinatorics, and Optimization

Dissertation Committee:
Willem-Jan van Hoeve (Chair)
Gérard P. Cornuéjols
John N. Hooker
Jeffrey T. Linderoth

Abstract

Mixed-integer programming (MIP) is often a practitioner's primary approach when tackling hard discrete optimization problems. This important role was enabled by decades of theory and practical experience poured into modern MIP solvers. However, many problems are still challenging for MIP solvers, which motivates the need for novel perspectives to enhance MIP technology.

In this dissertation, we explore the use of relaxed decision diagrams to improve MIP solvers. Relaxed decision diagrams are graph structures that encode relaxations of discrete optimization problems. One of their many uses in optimization is to generate bounds on the optimal value, which are strong in practice in the presence of certain types of structure.

However, exporting decision diagram techniques to integer programming can be challenging due to the lack of clear structure to exploit. A first step is to develop methods to construct good relaxed decision diagrams from integer programming models. We propose a framework that focuses on a substructure of the problem and incorporates remaining constraints via Lagrangian relaxation and constraint propagation. In particular, we explore the use of a prevalent substructure in MIP solvers known as the conflict graph. Computational experiments indicate that they yield strong bounds under this framework.

Once we understand how to build good relaxed decision diagrams, the next question is how to apply them in the context of a MIP solver. A MIP solver has several components, which include presolve, cutting planes, primal heuristics, and branch-and-bound. We show how relaxed decision diagrams can aid each of these components throughout the subsequent chapters.

In particular, we view decision diagrams from a polyhedral perspective in order to generate cutting planes. Through a connection between decision diagrams and polarity, we develop an algorithm that generates cuts that are facet-defining for the convex hull of a decision diagram relaxation. We provide computational evidence that this algorithm generates strong cuts for structured problems.

Finally, we investigate a broader integration of decision diagrams into MIP solvers. First, we show how relaxed decision diagrams can be used for bound and coefficient strengthening in the presolve step of a MIP solver. Second, we investigate the effect of applying cuts from decision diagrams throughout a branch-and-bound tree. Third, we generate both primal and dual bounds from decision diagrams to improve pruning within a branch-and-bound tree, which can result in significant improvements in solving time.

Acknowledgments

I am very glad to have taken the journey of obtaining a PhD alongside a number of amazing people who have supported, taught, and inspired me.

First and foremost, I would like to thank Willem-Jan van Hoeve for being an incredible advisor throughout these years. I have learned so much from Willem and I thoroughly enjoyed all the time we spent on whiteboards discussing and developing ideas together. Exploring untouched directions in research is often challenging and Willem's optimism and encouragement were crucial for me to push through. All the advice Willem has given me will be valuable for the rest of my career.

I would like to thank Gérard Cornuéjols, from whom I learned a lot from in an early research project and whose vision in research always inspires me. I am also grateful to François Margot and Giacomo Nannicini for teaching me the nuances of designing good computational experiments early on in my PhD – an important skill in computational research. Furthermore, I would like to thank Fatma Kılınç-Karzan for always being open to interesting conversations and offering me invaluable support and advice throughout my PhD.

I would like to thank the members of my committee (Gérard Cornuéjols, Willem-Jan van Hoeve, John Hooker, and Jeff Linderoth) for taking the time and effort to read this dissertation and provide interesting questions and comments. Beyond my main research at CMU, I am grateful to Kibaek Kim and Victor Zavala for hosting me for a research internship at Argonne, and to Srikumar Ramalingam for collaborating in an exciting and fruitful project.

I am grateful to Lawrence Rapp and Laila Lee, who are exceptional at taking care of all the administrative matters of the PhD program at Tepper and generally making our lives much easier.

I would like to thank all my colleagues in Tepper and ACO throughout all these years: Alex, Amin, André, Arash, Bo, Dabeen, Danial, David, Emilio, Franco, Gerdus, Jasper, Jenny, Jeremy, Joris, Leela, Mehmet, Michael, Nam, Neda, Negar, Ryo, Sagnik, Selva, Sercan, Sid, Stelios, Tarek, Thiago, Thomas, Wenting, Yang, and Ziye. This is a very smart group of people and it was a pleasure getting to know every one of them. I particularly thank those who have been closer to me; our friendships have helped me stay strong throughout these years. In addition, I am especially thankful for the extensive and fruitful research discussions with Alex and Thiago, some of which have positively influenced this dissertation.

Finally, I would like to thank my family, especially my parents. I would not have reached this point of my life without their support and encouragement.

Contents

Chapter 1

Introduction

In the past decades, the challenge of solving several of the discrete optimization problems encountered in practice has shifted from an algorithmic task to a modeling one. Modern mixed-integer linear programming (MIP) solvers have become remarkably powerful: hard problems that once required specialized algorithm development can now be tackled in practice with the simpler task of modeling them as integer programs and providing them to a solver. This current landscape has been shaped by several forces, including advances in computing power and parallelism. A great part of the reason is the development and implementation of effective generic algorithms within MIP solvers. From cutting planes to primal heuristics to branching rules, modern solvers aggregate several fine-tuned components that, when properly assembled together, form an efficient machine to solve discrete optimization problems.

Nevertheless, just as optimization technology progresses, so does the world. As MIP solvers put old problems behind, a surge of new, complex problems brings challenges to the table. Whether they come from the sharp increase in the scale of data sets we now need to handle, or from challenging requirements such as stochasticity or nonlinearity in which MIP solvers often play a key role, these problems need to be solved within reasonable time frames. Mixed-integer programming solvers must keep up with this new generation of optimization problems with the continued development and implementation of better algorithms. Incremental improvements, while important, may not suffice; we must seek fundamentally new ways to enhance integer programming solvers. This dissertation explores one of these new paths: the use of decision diagram relaxations within integer programming.

Decision diagrams (DDs) are data structures that can be used to represent discrete sets of points. In a nutshell, they are layered directed acyclic graphs in which arcs are associated to variable assignments and paths from a root node to a terminal node correspond one-to-one to the points of the represented set. In the context of optimization, these paths may encode the feasible solutions of a discrete linear optimization problem. Once such a decision diagram is built, we can efficiently find an optimal solution to the problem since it is equivalent to finding a maximum weighted path in a directed acyclic graph, which can be done efficiently. Thus, if we can encode a discrete optimization

problem with a small decision diagram, we can efficiently solve it. On the other hand, this means that in general we cannot expect to construct small decision diagrams for hard problems. In order to address this issue, decision diagrams can be used as approximations. In particular, we can construct tractable-sized decision diagrams that serve as relaxations for the problem: they include all feasible solutions and potentially some infeasible solutions. These are called relaxed decision diagrams.

This dissertation centers around the question of how to use relaxed decision diagrams to improve the performance of a MIP solver in practice. This can be naturally split into two research questions:

1. How to construct good relaxed decision diagrams for integer programming formulations?
2. Given good relaxed decision diagrams, how to use them to improve a MIP solver?

Good relaxed decision diagrams are those that have tractable size and approximate the problem well in some sense. How well the approximation is depends on how they are used: for instance, if they are used to generate objective bounds, we want the bounds to be close to the optimal value.

In this introduction, we set up the stage to answer these questions. We start with an overview of two perspectives we take in this dissertation. We next define basic concepts and discuss related work. Finally, we provide a description of each chapter in this dissertation.

1.1 Perspectives

Relaxed decision diagrams have shown to perform very well for specific types of problem classes, as we describe in the related work section later in this introduction. An underlying motivation in this dissertation is to enable DD-based techniques to be used effectively in wider contexts and applications than those currently in the literature. Examining decision diagrams under the lens of integer programming brings us closer to this goal.

We discuss two perspectives we take in this dissertation in order to answer the two research questions above. A first perspective relates to this goal of generality.

The main challenge that we have encountered in this research is that decision diagrams often rely on specific structure in order to stay tractable while providing good approximations. Due to this reason, it is difficult to make decision diagrams work well in practice for generic IP models. In fact, this was shown to be the case in much of our computational experience throughout the work leading to this dissertation. Later in this dissertation (Chapter 3) we illustrate this by comparing a generic approach with one that relies on structure.

Our solution to this challenge is not to make decision diagrams themselves more generic, but to embrace its reliance on structure within a generic setting. We do not seek to solve an arbitrary IP model; instead, we take the opportunistic approach to improve the solution process of instances that do have structure. This shift in perspective allows us to play to the strength of decision diagrams and avoid difficulties in generality when applied to inequality-based models. Our goal now becomes

to identify and leverage structure using relaxed decision diagrams within the generic setting of a MIP solver. In this dissertation, we show that this is indeed achievable through our computational experiments in Chapters 3 and 5.

A second perspective in this dissertation, tied more closely to the second question, is of representation and communication. The question of applying relaxed decision diagrams to integer programming can be fundamentally rephrased as follows: how to communicate information from one representation of the problem – a relaxed decision diagram – to another representation – an integer programming formulation? In other words, what inferences can we make from one representation to the other?

This parallels the concepts of $\mathcal{V}$-polyhedra and $\mathcal{H}$-polyhedra: vertex descriptions and hyperplane descriptions of polyhedra respectively. In essence, relaxed decision diagrams can be viewed as a particular encoding of a set of vertices of a polyhedron P that represents a relaxation of the problem, whereas the linear programming (LP) relaxation of the problem is given in terms of inequalities. Since P is fundamentally different than the LP relaxation of the problem, if we can infer inequalities that are valid for P, then we can potentially generate cutting planes and add them to the LP relaxation, which is essentially what we do in Chapter 4. In a way, we are bypassing the LP relaxation by building an alternative representation of the problem and communicating information back to the LP relaxation.

This of course relies on this alternative relaxation P being strong relative to the LP relaxation in some sense. We observe in our computational experiments that this can happen in the presence of structure, and thus strong cuts can be generated. The limitations of P are different from those of the LP relaxation – for instance, P can have exponentially many facets even if the underlying decision diagram is small. A simple example is the parity polytope, given by the convex hull of all binary solutions whose components sum up to an odd number. The number of facets of the parity polytope is exponentially large [41]. However, we can construct a decision diagram of width two because, as we build its decision diagram, there are only two states at each layer: either the components corresponding to assigned variables so far sum up to an even number or an odd number.

The converse direction is also important in this dissertation: how to infer information from linear constraints that are useful to decision diagrams. The compilation and refinement techniques described in Chapter 2 can be viewed as answers to this question. In addition, in Chapter 3, we propagate information from the linear constraints to the states embedded in the decision diagrams.

Finally, we do not necessarily need to alter either of the two representations in order to aid the MIP solver. We can aggregate information from relaxed decision diagrams and linear constraints using Lagrangian relaxation, allowing us to infer bounds on the optimal value. This is examined in Chapter 3.

1.2 Preliminaries

1.2.1 Integer programming

Integer programming is an area of optimization that encompasses a broad range of applications; see e.g. [33] for an in-depth treatment on the topic. We are generally interested in solving the following pure bounded integer linear programming problem:

$$\begin{aligned} \max\ & c^\top x \\ & Ax \leq b \\ & l \leq x \leq u \\ & x \in \mathbb{Z}^n \end{aligned} \tag{P}$$

Above, $c \in \mathbb{R}^n$ is a vector of objective coefficients, $l \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$ are lower and upper bounds for x respectively, and $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^m$ are the constraint coefficient matrix and right-hand sides respectively.

A *feasible solution* of (P) is one that satisfies the constraints of (P), or in other words belongs to the *feasible set*, $\{x \in \mathbb{Z}^n : Ax \leq b, l \leq x \leq u\}$. The *linear programming (LP) relaxation* of (P) is given by dropping the integrality constraints: $\max\{c^\top x : Ax \leq b, l \leq x \leq u\}$. The *integer hull* of (P) is the convex hull of the feasible set.

Given a variable x_j, we denote by $\mathcal{D}(x_j) := \{x_j \in \mathbb{Z} : l_j \leq x_j \leq u_j\}$ the *domain* of x_j, which corresponds to the set of values x_j can take if we ignore the constraints $Ax \leq b$. More generally, we let $\mathcal{D} := \{x \in \mathbb{Z}^n : l \leq x \leq u\}$ be the domains of the variables x, and thus (P) can be more succinctly expressed as $\max\{c^\top x : Ax \leq b, x \in \mathcal{D}\}$. Although $\mathcal{D}$ has this particular form in the context of an integer program, in fact throughout this dissertation we do not make any assumptions on $\mathcal{D}$ other than being discrete and finite.

We refer to any implementation that encompasses traditional techniques (described below) to solve a mixed-integer program as a *MIP solver*. Modern MIP solvers carry an extensive range of techniques; see [3] for an in-depth description of the open-source solver SCIP. We briefly outline its main mechanisms that are most relevant to this dissertation.

Before entering the bulk of the solving process, a MIP solver performs a *presolve* step, which includes a variety of techniques that make the problem easier to solve in the subsequent steps [4]. In the main process, a *branch-and-bound tree* is constructed as follows. At the root node of the tree, we solve the LP relaxation and obtain an optimal solution $\bar{x}$. We then select a variable x_j and branch to two children nodes, representing the subproblems with the bound constraints $x_j \leq \lfloor \bar{x}_j \rfloor$ and $x_j \geq \lceil \bar{x}_j \rceil$. We select a node according to some criteria and repeat this process, expanding the search tree. Whenever one of the subproblems yields an integer feasible solution, we keep it if it is better than the one currently stored, called an *incumbent solution*. Subproblems with infeasible LPs

can be discarded. After exploring the entire tree, we have an optimal solution to the problem.

This process is enhanced with bounds, primal heuristics, and cutting planes. The objective value of the incumbent solution is a *primal bound*, which is a lower bound if we are maximizing or an upper bound if we are minimizing. The LP relaxation at a node yields a *dual bound*, that is, an upper bound if we are maximizing or a lower bound if we are minimizing. If the primal bound is at least as good than the dual bound, then we know that there are no better feasible solutions at that node and we can prune (remove) it.

Primal heuristics are fast heuristics that search for good feasible solutions and are executed throughout the branch-and-bound tree. They not only provide primal bounds for the pruning process, but also allow for better feasible solutions as output if the process is terminated before reaching optimality.

The LP relaxations are strengthened with *cutting planes*, especially at the root node. A cutting plane is an inequality added to the formulation that cuts off a part of the LP relaxation without affecting the feasible set. This makes the LP closer to the integer hull.

We emphasize that throughout this dissertation we always make the assumption that the problem is pure integer and bounded (despite using the more general term "MIP solver").

1.2.2 Decision diagrams

A *decision diagram* (DD) can be viewed as a graph representation of a discrete set of points. An example is shown in Figure 1.1. In this definition, suppose that we are representing the feasible set S of a discrete optimization problem with variables $x_1, \dots, x_n$.

A decision diagram is a directed acyclic multigraph that has the following layered structure. The nodes of a decision diagram are partitioned into $n+1$ layers, where the first n layers correspond to the n variables in some fixed order, say $x_1, \dots, x_n$, and the last layer is reserved for a single terminal node t. The first layer contains a single root node s. Arcs are similarly assigned to layers: the layer of an arc corresponds to the layer of its tail. Each arc at a layer k is associated with an assignment of some $v \in \mathcal{D}(x_k)$ to the variable x_k. We call such an arc a v-arc and we say that v is its label. Parallel arcs are allowed; that is, there may be more than one arc between two nodes. We assume that every node except s and t has at least one incoming arc and at least one outgoing arc.

This structure represents S through the following property: there is a one-to-one correspondence between each point $x \in S$ and each directed path from s to t in the decision diagram. Given a directed path p from s to t, the corresponding x is such that $x_k = v_k$ where v_k is the label of the layer-k arc of p.

Note that a decision diagram is not unique for S. However, given a fixed variable ordering, a unique smallest decision diagram for S exists, called a *reduced decision diagram* [45].

In the case where the domains are binary, the decision diagram is a *binary decision diagram* (BDD); otherwise it is a *multivalued decision diagram* (MDD). Throughout this dissertation, we do

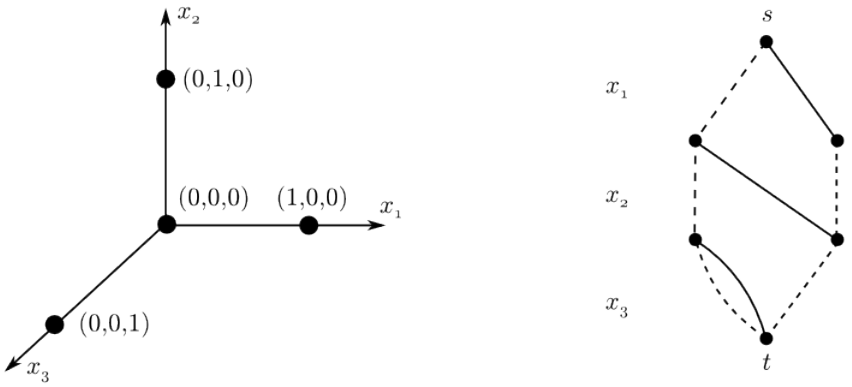


Figure 1.1: The decision diagram on the right, which has width 2, represents the set of four points on the left. A dashed line indicates a 0-arc and a full line indicates a 1-arc. Arcs are oriented from top to bottom. Note that each of the four paths from s to t in the decision diagram corresponds to a point in the set. For instance, following the rightmost arcs gives us the point $(1, 0, 0)$.

not impose particular domains except when specified.

A decision diagram may be enhanced with *long arcs* in order to reduce the number of nodes. In the definition of decision diagrams above, arcs go from a layer to the next. A long arc is an arc that skips layers; that is, it may go from a layer to any higher layer. Long arcs in general may represent any set of assignments to the variables between the two layers, indicated by their labels. A common use of long arcs is to replace nodes that have a single outgoing 0-arc. In this form, a long v-arc from layer k to $k + r$ represents the assignments $x_k = v, x_{k+1} = 0, \ldots, x_{k+r} = 0$. Such decision diagrams are called *zero-suppressed decision diagrams* (ZDDs) [49].

The traditional application of decision diagrams in the context of Boolean functions uses two terminal nodes, representing the "true" and "false" evaluation of the function, respectively. In our context it suffices to use only the "true" terminal node to represent all points in S.

In the context of optimization, a very useful property of decision diagrams is that, once built, they can be used to optimize any additively separable objective function – such as linear functions – over the set of points they represent. This can be done as follows: suppose that we want to maximize $\sum_{i=1}^{n} f_i(x_i)$. If we assign weights $f_i(v)$ to every v-arc of layer i, then each maximum weighted path of the decision diagram represents an optimal solution, due to the one-to-one correspondence between paths and feasible solutions. Given that a decision diagram is a directed acyclic graph, finding an optimal path can be done in time linear in the number of arcs of the decision diagram.

However, it is typically not efficient to have a decision diagram represent its feasible set exactly, since it may have exponentially many nodes with respect to the size of the problem. As an

approximation, we consider tractable decision diagrams that are only required to contain the feasible set and not represent it exactly. These decision diagrams are called *relaxed decision diagrams* [17, 20] and are usually of size polynomial or linear in the size of the problem. Formally, given a set of points $S \subseteq \mathcal{D}$, a relaxed decision diagram with respect to S is a decision diagram that represents a set S' that contains S. To differentiate relaxed decision diagrams from decision diagrams representing exactly the set of feasible points, we call the latter *exact decision diagrams*.

The general idea of constructing a relaxed decision diagram is to restrict the number of nodes of its largest layer, called *width*, during its construction in such a way that no feasible points are removed in the process. We discuss this construction further in the next chapter.

One of the ways that a relaxed decision diagram can be useful is that if we optimize the objective function over it, we obtain a dual bound for the problem. In particular, if the width of this decision diagram is constant, then the bound is obtained in constant time.

The following additional concepts are useful for the compilation and manipulation of decision diagrams. Given a node u of a decision diagram, a *partial solution* of u is a path from the root s to u, and a *completion* of u is a path from u to the terminal t. For convenience, we use the same notation to denote the corresponding points: for some k, a partial solution is a point in the space of the first k variables, and a completion is a point in the space of the last k variables. We define the *partial solution set* $S^{\uparrow}(u)$ and the *completion set* $S^{\downarrow}(u)$ of a node u as the set of all partial solutions of u and completions of u respectively.

Throughout this dissertation, we make some simplifications in notation for convenience. We denote an s-t path in a decision diagram by simply a path, unless the context identifies its start and end nodes. Moreover, due to the one-to-one correspondence, we often treat a path as a solution and vice versa. In particular, we call a path feasible or infeasible if the solution it represents is feasible or infeasible respectively with respect to the problem or constraint being considered.

1.2.3 Related work

Decision diagrams are very flexible structures due to their simplicity, and thus appear in several contexts. They were originally proposed in the context of circuit design and formal verification [5, 26, 47].

In optimization, they have been used for constraint programming [6, 39, 37, 15], bound generation [17, 20, 22], multiobjective optimization [14, 21], postoptimality analysis [54], nonlinear optimization [13, 34], and other applications. Decision diagrams have also been used in the context of integer programming, which includes cut generation with exact decision diagrams [11] (a different method than proposed in this dissertation, as we discuss in Chapter 4) and a branching-based approach [46].

Decision diagrams are also effective in a number of practical applications, such as sequencing problems (which include scheduling and routing problems) [32, 44], portfolio optimization [13], and

graph problems such as maximum independent set [22].

Many of these applications and techniques are summarized in [19].

1.3 Overview of Dissertation

The contents of this dissertation are summarized next.

Chapter 2: Decision Diagram Compilation and Refinement

Chapter 2 is devoted to laying out the tools for constructing and refining decision diagrams. We contrast two compilation approaches, top-down and depth-first, and describe compilation for linear constraints. On refinement, we discuss arc filtering and an extension based on bounds we call arc pruning. We summarize complexity results related to compilation and refinement. Most of the algorithms and results in this chapter come from prior work.

Chapter 3: Decision Diagram Relaxations for Integer Programming Models

In Chapter 3, we address the question of how to construct an effective relaxed decision diagram from an integer programming model. We propose a framework that builds a decision diagram based on a substructure of the problem and incorporates the remaining constraints via Lagrangian relaxation and constraint propagation. We show how to efficiently construct a reduced decision diagram for the conflict graph, a structure present in modern MIP solvers. Bounds from this framework are compared with LP bounds and bounds from direct compilation from linear constraints on instances that combine set packing and knapsack constraints.

Chapter 4: Cutting Planes from Relaxed Decision Diagrams

Chapter 4 takes a polyhedral view of decision diagrams. We develop an algorithm to generate strong cutting planes from decision diagrams called target cuts. They are facet-defining with respect to the convex hull of the relaxed decision diagram being considered. In addition, we show how to certify a lower bound on their dimension with respect to the integer hull of the problem. These cuts are tested on the maximum independent set problem and the minimum set covering problem.

Chapter 5: Integrating Decision Diagrams into MIP Solving

In Chapter 5, we investigate further approaches to integrate decision diagrams into MIP solvers. We discuss bound and coefficient strengthening from presolve from the perspective of decision diagrams. Next, we computationally experiment with the embedding of cutting planes, primal bounds, and dual bounds from decision diagrams throughout the branch-and-bound tree.

All contribution in this dissertation is joint work with Willem-Jan van Hoeve.

Chapter 2

Decision Diagram Compilation and Refinement

2.1 Introduction

In this chapter, we describe various tools to construct and manipulate decision diagrams for purposes of optimization. The aim of these tools is to construct a decision diagram that approximates well the feasible set of a problem – in some cases taking into account the objective function as well. Knowing these methods is crucial to understanding the effectiveness and limitations of decision diagram-based approaches in optimization.

We focus exclusively on decision diagrams aimed at representing the feasible set of a discrete optimization problem, exactly or approximately. It is also possible to use decision diagrams to represent objective functions instead [13], but this application is not within our scope.

These tools can be organized into two types: compilation and refinement.

- *Compilation techniques* construct a decision diagram from scratch, typically using a dynamic programming formulation. We compare two common compilation approaches: top-down and depth-first.

- *Refinement techniques* are used to improve upon an existing decision diagram in order to make it a better approximation of the desired feasible set or reduce its size. This typically consists of removing infeasible solutions, but may also include removing feasible suboptimal solutions or adding solutions that result in merging nodes. These techniques include arc filtering, node splitting, and constraint separation. In addition, we present arc pruning, a version of arc filtering that is performed at construction time.

Given that we use decision diagrams in the context of integer linear programming, we illustrate the application of these techniques for linear inequalities in this chapter. A different approach

that relies on problem substructure within an integer programming model is discussed in the next chapter.

2.2 Dynamic Programming Formulations

Decision diagrams are often compiled from dynamic programming (DP) formulations. Dynamic programming is a well-known recursive approach to solve optimization problems.

In DP, a problem is defined in terms of *states*, a *transition function*, and a *transition cost function*. Each state represents a subproblem to be solved and the transition function corresponds to a decision that reduces a subproblem to another one while paying a cost given by the cost function. A state typically has multiple transitions associated to different values assigned to a decision variable. The solution process starts at the root state, which represents the original problem, and performs transitions until a terminal state is reached. In a traditional version of DP, this is performed exhaustively for all possible transitions. The optimal solution corresponds to the set of transitions with minimum cost (assuming minimization).

A common way to visualize a DP is in terms of its state transition graph, in which nodes are states and labeled arcs correspond to transitions. In fact, a decision diagram can be interpreted as a state transition graph, as seen in Figure 2.1.

The following definition of a DP formulation is slightly tailored to our purposes. In particular, we represent implicitly in the DP model an ordered set of variables $x_1, \dots, x_n$ with domains $\mathcal{D}_1, \dots, \mathcal{D}_n$ respectively. Transitions correspond to assignments of variables. The state space is divided into *stages*. A state in a stage j is associated to a subproblem in which the variables $x_1, \dots, x_{j-1}$ have been assigned and $x_j, \dots, x_n$ are yet to be assigned. While we fix the variable ordering, we require that the DP formulation is defined for every possible ordering of variables.

We define a DP formulation for a problem $\mathcal{P}$ with n variables as follows:

States For each $j = 1, \dots, n+1$, a state space $\mathcal{S}^j$ of stage j consists of states s, which can take any form. States in different stages are never considered equivalent, even if they would be otherwise.

Special states are the root state $\hat{r}$, which represents $\mathcal{P}$ in its entirety, a feasible terminal state $\hat{1}$, and an infeasible terminal state $\hat{0}$. We have $\mathcal{S}^1 = \{\hat{r}\}$ and $\mathcal{S}^{n+1} = \{\hat{0}, \hat{1}\}$. The infeasible terminal state $\hat{0}$ is replicated in all stages; that is, $\hat{0} \in \mathcal{S}^j$ for all $j = 1, \dots, n+1$.

The state space $\mathcal{S}$ of the DP formulation is the union of all state spaces of each stage $\mathcal{S}^j$.

Transition function For each $j = 1, \dots, n$, the transition function $t_j : \mathcal{S}^j \times \mathcal{D}_j \to \mathcal{S}^{j+1}$ provides the state s' representing the result of the assignment of an input value $v_j \in \mathcal{D}_j$ to the variable x_j at an input state s.

Conventional definitions of a DP formulation may include a transition cost function and additional terminal states. The reason we omit the transition cost function from our DP models is because we are mainly concerned with modeling only the feasible set of constraints.

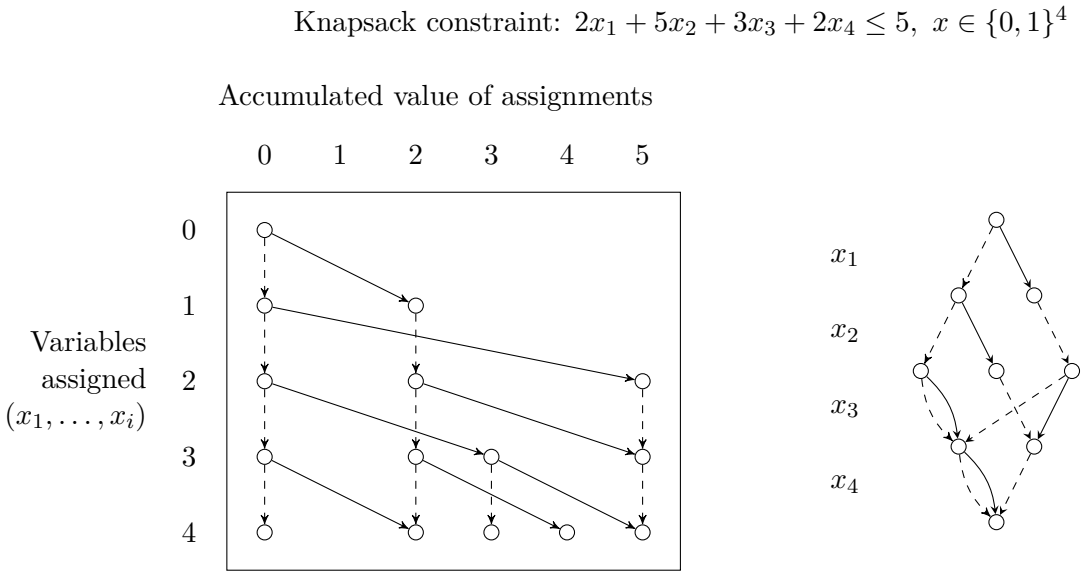


Figure 2.1: On the left, the state transition graph for a binary knapsack constraint. A typical DP formulation of a knapsack constraint has as states (i, W), where i corresponds to having assigned variables $x_1, \ldots, x_i$ and W is the accumulated value from these assignments in the left-hand side expression. The transition function updates W according to the assignment, ensuring it does not exceed the right-hand side. On the right, a reduced decision diagram of the same constraint. In both of them, dashed and solid arcs correspond to assigning zero and one respectively.

2.3 Decision Diagram Compilation

In essence, constructing a decision diagram from a DP formulation consists of exploring its state space. Starting at the root state, we iteratively take an unvisited state and generate the states it transitions to until all states have been visited. We build the decision diagram along with this process, establishing states as nodes and transitions as arcs. The transitions follow a given variable ordering. If an existing state is reached through a different set of transitions from the root, the same node must be used. In fact, these shared states are fundamental for the compactness of a decision diagram; without them, the decision diagram would become a branching tree. If the infeasible state $\hat{0}$ is reached, we discard the associated node. Algorithm 2.1 details this procedure. Nodes that are yet to be branched on are called *open*, and those which have been are denoted by *explored* (or *visited*).

Algorithm 2.1 Framework for decision diagram compilation

Input: Dynamic programming formulation with state set $\mathcal{S}$, root state $\hat{r}$, and transition functions t_j for variable x_j, $j = 1, \ldots, n$, variable ordering σ where $\sigma(j)$ is the layer of variable x_j
Output: Decision diagram D corresponding to the feasible set of the formulation

DD-Compilation($\mathcal{S}, \hat{r}, t_j$)

Initialize decision diagram D
$r \leftarrow$ root node in D with state $\hat{r}$
$O \leftarrow \{r\}$ ▷ Set of open nodes
Initialize hash maps P_j for all $j = 1, \ldots, n$ ▷ State pool for each layer j
$P_1[\hat{r}] \leftarrow r$
while $O \neq \varnothing$ **do**
  $u \leftarrow$ SelectNode(O)
  $s \leftarrow$ state attached to u
  $j \leftarrow$ index of variable corresponding to u
  for all v in domain $D_s(x_j)$ **do**
    $s_{\text{new}} \leftarrow t_j(s, v)$
    if s_{new} is feasible **then**
      if $s_{\text{new}} \in P_{\sigma(j)}$ **then**
        $u_{\text{new}} \leftarrow P_{\sigma(j)}[s_{\text{new}}]$
      else
        $u_{\text{new}} \leftarrow$ new node in D with state s_{new}
        $O \leftarrow O \cup \{u_{\text{new}}\}$ if u_{new} is not terminal
      Add arc (u, u_{new}) to D with value v
return D

The order in which we explore nodes, given by SelectNode in Algorithm 2.1, can make a significant difference. We highlight two main approaches: a top-down construction and a depth-first construction. Differences between these two approaches are detailed in Section 2.6.

In a top-down construction, SelectNode chooses a node in the smallest layer first. It may sometimes be called a breadth-first construction, but if there are long arcs it may not follow the typical breadth-first search. It has the property that, at any point, any explored node has its partial solution set completely explored. This is a consequence of the fact that, if we have explored a node, then we must have explored all nodes in previous layers.

In a depth-first construction, SelectNode chooses a node of largest depth. That is, we explore all descendants of a node before moving to a sibling. Typically, the set of open nodes is implemented as a stack: the last node added to it is the next node to be evaluated. We call a node *fully explored* if all of its descendants are explored, or equivalently if its completion set is fully constructed. It has the property that, whenever we visit an open node, all nodes of the same layer or higher are either fully explored or open.

The node selection function does not need to be restricted to these two approaches. It can be useful to let SelectNode be driven by a heuristic, based on some measure of how promising a node is. This search strategy has been applied for instance in bin packing problems [42]. Furthermore, node selection strategies from other contexts such as MIP solving may be applied here, although goals of the procedure and available information may differ.

Long arcs can be implemented by delaying the consolidation of an open node. For instance, consider the case of long arcs of the form $(*, 0, \dots, 0)$ for a binary problem, where $*$ can take 0 or 1. At the beginning of the evaluation of an open node, we check if the only feasible assignment is of value zero. If so, we prepare it to be evaluated in the next layer, updating it with a new state if necessary, and keep it in the list of open nodes.

2.4 Equivalence Tests

Although Algorithm 2.1 correctly produces a decision diagram representing the given DP formulation, it may not be the most compact one. While certain DP formulations inherently yield reduced decision diagrams, it is often the case that the resulting decision diagram is not reduced. In order to have reduced decision diagrams, any two nodes that have the same completion set – which we call *equivalent* – must be merged into one.

During construction however, the completion sets of both nodes are typically not available. Instead, we have states. Concepts of completion sets and equivalence for nodes are analogously defined for states as follows. The *completion set of a state* is the set of completions associated with the sets of transitions from the state to the feasible terminal state. Likewise, *equivalent states* are those with the same completion set.

Within the compilation of a decision diagram, an *equivalence test* decides whether to merge

two nodes or not. In order for the decision diagram to correctly represent the DP formulation, the equivalence test must be *correct* (or *sound*), which means that it does not merge nodes with non-equivalent states. Assuming the equivalence test is correct, the merged node can take either state. The equivalence test is *complete* if it always merges nodes whenever their states are equivalent. If we can efficiently perform complete equivalence, then we can efficiently construct a reduced decision diagram without exploring more nodes than those in the output.

We denote by the *complete equivalence problem* the problem of deciding if two states are equivalent. We make a distinction between the complete equivalence problem in a top-down compilation and in a depth-first one. In a top-down compilation, the input is the pair of states. In a depth-first compilation, we attempt to merge an open node with a fully explored node. Thus, in addition to the states, the input includes the completion set of the fully explored node in the form of a decision diagram. Due to this extra input, complete equivalence in depth-first is no harder than in top-down.

Nevertheless, complete equivalence tests help us avoid revisiting occurrences of equivalent nodes, and thus it is useful to delineate when they can be done efficiently. As mentioned in [37] and established below, deciding complete equivalence is as hard as deciding feasibility.

Proposition 2.1. *Let $\mathcal{P}$ be a problem or constraint set with discrete domains. Suppose an infeasible instance for $\mathcal{P}$ exists and is given. If it is NP-complete to decide feasibility for $\mathcal{P}$, then it is NP-complete to decide complete equivalence for $\mathcal{P}$, whether in a top-down or a depth-first compilation.*

Proof. Let s_1 and s_2 be the root states of an arbitrary instance I for $\mathcal{P}$ and an infeasible one respectively. In the depth-first case, we let s_1 be the fully explored node, which is equivalent to only providing the states as input. Deciding complete equivalence between s_1 and s_2 is equivalent to the problem of deciding feasibility of I. Therefore, if feasibility is NP-complete, then complete equivalence is also NP-complete. □

The converse is not true, as illustrated by the alldifferent constraint, which ensures that all variables take different values from their domains. It is easy to determine feasibility for the alldifferent constraint, but deciding complete equivalence for it either in a top-down or depth-first compilation is NP-complete (see Section 2.9).

Proposition 2.1 indicates that complete equivalence is NP-complete for several common constraints, such as 3-SAT and (single) linear equalities. However, in practice, efficiently deciding complete equivalence may not always be necessary to construct compact decision diagrams. In some cases, it may be enough in practice to find a sufficient condition for equivalence that is frequently satisfied.

Moreover, we can always reduce a decision diagram after construction with a bottom-up pass [26]. In short, the reduction process works by traversing each layer from the bottom to the top and merging every set of nodes with exactly the same children. Performing reduction (and other operations on decision diagrams) can be parallelized [50].

2.5 Relaxed Decision Diagram Compilation

The framework described in the previous section applies to exact decision diagrams. Often, exact decision diagrams can be exponentially large with respect to the size of the problem or constraint. This includes the case of a single linear inequality over binary domains [40]. In fact, since optimizing over a decision diagram can be done in time linear to its size, any optimization problem that admits a decision diagram that can be constructed in polynomial time must be polynomially solvable. As we are interested in solving NP-hard problems, constructing exact decision diagrams for a problem in its entirety does not scale well for our purposes.

In order to make a decision diagram tractable, we can represent an approximation of the decision diagram instead. The compactness of a decision diagram comes from merging equivalent nodes. We can make decision diagrams smaller by merging non-equivalent nodes, which if done carefully, allows us to construct a restriction or a relaxation of the problem. Given that in this dissertation we focus on relaxations, we describe how to construct a relaxed decision diagram in this section. We assume a top-down compilation.

To obtain a relaxation, merging non-equivalent nodes must not remove feasible solutions from the decision diagram. It requires a problem-specific *merging operator* that takes two states and returns a state representing a relaxation of both. More precisely, a merging operator $\oplus : \mathcal{S}^j \times \mathcal{S}^j \to \mathcal{S}^j$ takes two states s_1 and s_2 in the same stage j and returns a state s such that $S^{\downarrow}(s) \supseteq S^{\downarrow}(s_1) \cup S^{\downarrow}(s_2)$. This guarantees that no feasible solution is lost in the process.

In terms of the construction algorithm, the only difference between compiling an exact decision diagram and a relaxed one is that at certain points of the procedure we merge non-equivalent nodes using a given merging operator. In the traditional approach for constructing relaxed decision diagrams [6], this merging process is performed in a systematic way. We provide to the algorithm a width parameter W, which defines the maximum allowed width of the decision diagram. At the end of every layer, if its width exceeds W, then we merge enough nodes in order to make it at most W. This guarantees that a resulting decision diagram with n layers has size limited by $O(nW)$. There are several different heuristics to merge nodes, which may be problem-specific or generic.

2.6 Comparison between Top-Down and Depth-First Compilations

The order in which we explore nodes may affect the size of the final decision diagram and our ability to construct relaxations. The choice of approach depends on the purpose of the decision diagram. In subsequent chapters of this dissertation, we focus on the top-down construction, for which relaxations are more suitable.

2.6.1 Node equivalence

In a top-down construction, node equivalence is always checked between open nodes. While we know their states, we do not have information about the completion set of either node at this step. In a depth-first construction however, whenever we reach a new node and seek an equivalent node to merge with, all candidates have their completion sets fully constructed in the form of decision diagrams.

For certain problems such as the maximum independent set problem, this distinction is not important: complete equivalence can be efficiently performed in a top-down construction [22]. In other words, state information is sufficient for complete equivalence.

In the case of a single linear inequality however, complete equivalence is NP-complete in a top-down construction, but can be done in polynomial time in a depth-first construction. The completion set of a node allows us to compute its *equivalence class*, defined as the set of states that lead to the same completion set. We discuss this procedure in Section 2.7.1; see also [1, 12].

There are also cases such as multiple linear constraints in which complete equivalence is NP-complete in both top-down and depth-first compilations, as we discuss in Section 2.7.2.

2.6.2 Merge-based relaxations

Top-down constructions are considerably more suitable for the merge-based relaxation described in Section 2.5. Whenever we visit a node, all other nodes in the same layer are still open, allowing us to merge nodes by simply updating states. On the other hand, if we want to merge nodes in a depth-first compilation, we may need to update the descendants of one of the nodes in order to avoid losing feasible solutions, which may be expensive.

Furthermore, having complete layers before merging allows us to make more informed choices in merging. Not only we have all merging candidates available if we perform the merging step at the end of each layer, but also we have the full partial solution set of each candidate, which can guide the choice of nodes to merge. For instance, if our goal with the relaxation is to produce strong dual bounds, an effective heuristic is to merge nodes associated to partial solutions with small objective values (assuming maximization), since we want any added infeasible solutions to have small objective value in order to keep the bound tight [22].

2.6.3 Memory usage

In certain applications, states can occupy a substantial amount of memory and it is desirable to free that memory if the state is not used anymore. For instance, in the case of multiple linear constraints, the state has size $O(m)$, where m is the number of linear constraints. In the maximum independent set problem, the state has size $O(n)$, where n is the number of variables. By the end of the construction of a decision diagram, if we do not free any state from memory, the contribution

of the states to the space consumption is $O(NS)$, where N is the number of nodes of the decision diagram and S is the state size.

Assume that we free a state from memory when its corresponding node is no longer a candidate for equivalence – that is, the node is explored and there are no open nodes in earlier layers. In a top-down compilation, it is possible to free the states of layers that have been already processed. This reduces the space complexity of states to $O(WS)$, where W is the width of the decision diagram. However, in a depth-first compilation, we may need to keep many more states in memory. For instance, suppose we have completely explored one of the root's two children. In general, we need to keep all states except the root in memory since the second children and its descendants may be equivalent to the existing nodes. This is true even if we end up immediately merging the second children, which terminates the process with almost all states in memory.

2.6.4 Variable ordering

Variable ordering can have a substantial effect on the size of the decision diagram [26]. A depth-first compilation forces us to define the variable ordering essentially a priori. In a top-down compilation however, we can choose the variable ordering dynamically. Information from a complete layer can be helpful to decide the next variable to branch on. For instance, the variable ordering heuristic we use in Section 3.3.2 selects the variable that results in the smallest number of arcs in the following layer.

2.7 Decision Diagram Compilation for Linear Inequalities

The methods in this dissertation focus on leveraging specific structures within an integer program. Information about structure allows us to construct better decision diagrams. Nevertheless, it is also possible and in some cases practical to construct decision diagrams directly from linear inequalities. Even in cases where this is not practical, understanding where the roadblocks are in this natural approach can guide us in identifying a better one.

In this section, our domain $\mathcal{D}$ is given by $\{x \in \mathbb{Z}^n : l \leq x \leq u\}$, where l and u are the lower and upper bound vectors of the variables. We first consider the case where we want to represent a single linear inequality before moving on to the case of multiple linear inequalities. We consider mainly the exact construction in the following subsections. After the exact case, we define the merging operator for the relaxed case.

2.7.1 Single linear inequality

Dynamic Programming Formulation

A single linear inequality can be modeled with a DP that essentially keeps track of the resulting inequality after assigning values to variables. For instance, if we assign $x_1 = 1$ to the constraint $2x_1 + 3x_2 - 3x_3 \leq 3$, the constraint becomes $3x_2 - 3x_3 \leq 1$, assuming we move the new constant

term of 2 to the right-hand side. However, we do not need to store the entire constraint: within the same layer, these constraints have the same left-hand side because the set of assigned variables is the same. Therefore, all we need to store as a state is the right-hand side of the subproblem inequality (or alternatively, the value accumulated by the variable assignments). In the above example, the initial constraint corresponds to a state of 3, which turns into 1 after the assignment. At the last layer, we check the feasibility of the final constraint. In essence, this is the same DP as the well-known DP formulation for the knapsack problem (see e.g. [43]).

More precisely, a DP formulation for a single linear inequality $a^\top x \leq b$ can be defined as follows:

State space. For each stage j, the state space $\mathcal{S}^j$ contains values $R \in \mathbb{R}$ representing the right-hand side of an inequality. The root state $\hat{r}$ is b.

Transition function. For $j = 1, \ldots, n$,

$$t_j(R, v_j) = R - a_j v_j$$

If $j = n$, as an additional step the output R' of the above function must be converted into either a feasible or infeasible state. If $R' \geq 0$, then it becomes the feasible state $\hat{1}$. Otherwise, it is converted into the infeasible state $\hat{0}$.

This DP formulation correctly models the feasible set of the linear constraint because each state exactly represents the linear constraint after assignments are made. However, this formulation may not lead to complete equivalence. In other words, there are nodes with the same completion set that are not merged together. A question is when complete equivalence can be efficiently performed, which depends on whether we are doing a top-down compilation or a depth-first one.

Top-down compilation

Deciding complete equivalence for a linear inequality when we only have state information, such as in a top-down compilation, is NP-complete.

Proposition 2.2. *Given the above DP formulation for a single linear inequality, it is NP-complete to perform a complete equivalence test in the context of a top-down compilation, even when domains are binary.*

Proof. Consider two states with right-hand sides R_1 and R_2 and assume $R_1 \leq R_2$ without loss of generality. Assume that the domains are binary. These states have the same completion set if and only if the set $\{x \in \{0,1\}^n : R_1 < a^\top x \leq R_2\}$ is empty. Deciding whether this set is empty is a generalization of the subset sum problem, which can be viewed as determining whether $\{x \in \{0,1\}^n : -\epsilon < a^\top x \leq 0\}$ is empty for a sufficiently small ϵ. Since the subset sum problem is NP-complete, determining complete equivalence is also NP-complete. □

Despite this negative result, there are two cases in which complete equivalence test can be efficiently performed given only states as input: when the state corresponds to the infeasible state and when it corresponds to the state containing all feasible solutions in $\mathcal{D}$ [37]. In particular, the constraint is violated by all $x \in \mathcal{D}$ if and only if $\min_{x\in\mathcal{D}}\{a^\top x\} > b$. Moreover, the constraint is satisfied by all $x \in \mathcal{D}$ if and only if $\max_{x\in\mathcal{D}}\{a^\top x\} \leq b$.

In order to cover these two cases, we can extend the DP formulation as follows. For each layer l, precompute $L_l := \min_{x^\downarrow\in\mathcal{D}^\downarrow}\{a^{\downarrow\top}x^\downarrow\}$ and $U_l := \max_{x^\downarrow\in\mathcal{D}^\downarrow}\{a^{\downarrow\top}x^\downarrow\}$, where $a^\downarrow$ and $\mathcal{D}^\downarrow$ correspond to a and $\mathcal{D}$ projected onto the completion variables at layer l. Every time we find a state R such that $L_l > R$, we let the transition function convert it to the infeasible state $\hat{0}$ and the associated node is removed from the decision diagram. When $U_l \leq R$, the transition function converts it to a (possibly new) layer-l state that can only reach the feasible terminal state $\hat{1}$.

A common case is when all coefficients and variables are nonnegative and variable domains include zero. In this case, L_l is always zero and it suffices to check if a state is negative in order to determine it is infeasible.

Depth-first compilation

If we use depth-first compilation instead, complete equivalence for a single linear inequality can be performed in polynomial time. A reduced decision diagram can be constructed in $O(N \log W)$ time using depth-first construction, where N is the number of nodes of the decision diagram and W is its width, assuming that domains have constant size [12, 1].

In this setting, all candidates for merging have their completion set fully constructed. From this structure, we can compute equivalence classes at each node, which consist of sets of all right-hand sides that lead to the same completion set. In the case of a single linear inequality, equivalence classes are intervals $[L, U)$ for some $L, U \in \mathbb{R}$. The problem of merging equivalent nodes is then reduced to finding the equivalence class associated to the current state.

We recast the depth-first compilation algorithm from [12, 1] into a more general framework. We consider the construction of the feasible set of $\{x \in \mathcal{D} : f(x) \leq b\}$ for general functions f. When f is linear, we have the single linear inequality case.

The basis of the algorithm can be summarized in the following simple lemma. It states that a set of points is exactly the feasible set of a constraint if and only if its right-hand side lies within a certain interval. The lemma by itself is unrelated to decision diagrams, but it hints at using these intervals to determine equivalence within a decision diagram. In the lemma, we make the usual assumption that maximizing or minimizing over an empty set results in $-\infty$ or $+\infty$ respectively.

Lemma 2.3. *Let $f : \mathcal{D} \to \mathbb{R}$ be a function and $S \subseteq \mathcal{D}$ be a set of points. Then*

$$S = \{x \in \mathcal{D} : f(x) \leq b\} \iff \max\{f(x) : x \in S\} \leq b < \min\{f(x) : x \in \mathcal{D} \setminus S\}.$$

Proof. The following equivalences hold:

$$S \subseteq \{x \in \mathcal{D} : f(x) \leq b\} \iff \forall x \in S, f(x) \leq b \iff \max\{f(x) : x \in S\} \leq b,$$

$$\begin{aligned} S \supseteq \{x \in \mathcal{D} : f(x) \leq b\} &\iff \mathcal{D} \setminus S \subseteq \{x \in \mathcal{D} : f(x) > b\} \\ &\iff \forall x \in \mathcal{D} \setminus S, f(x) > b \\ &\iff \min\{f(x) : x \in \mathcal{D} \setminus S\} > b. \end{aligned}$$

□

We emphasize that the Lemma 2.3 holds for any real-valued function f as long as it is defined in the domain $\mathcal{D}$. In fact, the conditions are indifferent to any value $f(x)$ outside $x \in \mathcal{D}$.

In the context of decision diagrams, Lemma 2.3 can be used to determine equivalence between a node that has just been created, with a state representing $\{x \in \mathcal{D} : f(x) \leq b\}$, and a node with a constructed completion set S.

Consider the case where the function is linear: $f(x) = a^\top x$. Since we maintain the constant term in the right-hand side, the left-hand side is always the same across all nodes in a given layer. Suppose that we have stored in every fully explored node with a completion set $S^\downarrow$ over domains $\mathcal{D}^\downarrow$ the interval $[L_{S^\downarrow}, U_{S^\downarrow})$, where $L_{S^\downarrow} := \max\{a^{\downarrow\top} x^\downarrow : x^\downarrow \in S^\downarrow\}$, $U_{S^\downarrow} := \min\{a^{\downarrow\top} x^\downarrow : x^\downarrow \in \mathcal{D}^\downarrow \setminus S^\downarrow\}$, and $a^\downarrow$ is a projected onto the space of the completion variables of the node. By Lemma 2.3, a state with value R is equivalent to the node carrying the equivalence class $[L_{S^\downarrow}, U_{S^\downarrow})$ that contains R. By keeping these intervals in increasing order, finding such a node can be done via binary search on all nodes of the layer in time $O(\log W)$.

It is left to detail how $L_{S^\downarrow}$ and $U_{S^\downarrow}$ are computed. Since f is linear, these values for a node u can be computed with a bottom-up pass from the terminal to u. We do not require a full pass for every node: as soon as we identify that node u is fully explored, we know that its children are also fully explored and thus we can use their intervals to compute the interval of node u. The interval of the feasible terminal node is $[0, +\infty)$, as the left-hand side becomes zero after all variables are assigned.

Moreover, arcs that were removed due to infeasibility must be taken into account. For purposes of computing intervals, we add back these arcs, pointing to artificial nodes with infeasible equivalence classes in the following layer. In top-down compilation, we have seen that the infeasible equivalence class at layer l is given by $(-\infty, U_l)$, where $U_l := \min\{a^{\downarrow\top} x^\downarrow : x^\downarrow \in \mathcal{D}^\downarrow\}$. At the last layer, this interval is $(-\infty, 0)$. In terms of implementation, adding infeasible arcs does not need to be done explicitly and can be treated as a special case when computing intervals.

This exact same algorithm works if the function is additively separable: $f(x) = \sum_{i=1}^n g(x_i)$ for some function $g : \mathcal{D} \to \mathbb{R}$.

When $f(x)$ is not additively separable, nodes in the same layer may have different left-hand

sides. For instance, if we have a term x_1x_2 in $f(x)$, setting x_1 to 0 eliminates the term, while setting x_1 to 1 keeps a term of x_2 in the left-hand side. Nevertheless, Lemma 2.3 still induces an algorithm for complete equivalence. In particular, suppose that we can maximize and minimize any $g(x)$ in polynomial time over a decision diagram, where $g(x)$ is $f(x)$ with a subset of variables assigned. Assume also that we store left-hand side information in the states. Then given two states s_1 and s_2 and a completion set $S^{\downarrow}$ of s_1, we can compute an interval for s_1 by optimizing with respect to the left-hand side of s_2 over $S^{\downarrow}$, and thus by Lemma 2.3 decide complete equivalence in polynomial time.

Conversely, if optimizing over a decision diagram is NP-hard, then constructing an interval is NP-hard, since the lower bound of the interval is an optimization problem. This also applies if optimizing over $\mathcal{D}$ is NP-hard since a reduced decision diagram representing $\mathcal{D}$ has width one. This is the case, for example, with binary quadratic functions.

We remark that this depth-first compilation with equivalence classes is only useful in practice when there are several different possible states that would fall into the same equivalence class. For instance, this can happen when the coefficients and right-hand side are very different from each other. When the coefficients and right-hand side are similar to each other and small, this is less likely to happen and therefore computing equivalence classes is less beneficial.

2.7.2 Multiple linear inequalities

In the context of integer programming, we are typically interested in constructing decision diagrams for several linear inequalities. The DP formulation for a single linear inequality can be naturally extended to multiple linear inequalities. Instead of a single right-hand side, we store the right-hand sides of all inequalities in each state.

A DP formulation for a system of m linear inequalities $Ax \leq b$ with n variables is defined as follows:

State space. For each stage j, the state space $\mathcal{S}^j$ contains vectors $r = (R_1, \dots, R_m) \in \mathbb{R}^m$ representing the right-hand sides of the m inequalities. The root state is the vector b.

Transition function. For $j = 1, \dots, n$,

$$t_j(r, v_j) = r - a^j v_j$$

where a^j corresponds to the column j of A.

If $j = n$, as an additional step the output r' of the above function must be converted into either a feasible or infeasible state. If $r'_i \geq 0$ for all $i = 1, \dots, m$, then it becomes the feasible state $\hat{1}$. Otherwise, it is converted into the infeasible state $\hat{0}$.

However, in this case, complete equivalence becomes difficult, whether in a top-down or a depth-first compilation.

Corollary 2.4. *Deciding complete equivalence for two or more linear constraints in NP-complete, whether in a top-down or a depth-first compilation, even when variables are binary.*

Proof. Proposition 2.1 states that if feasibility is hard, then complete equivalence is hard. Since deciding 0-1 feasibility for two or more linear constraints is NP-complete, deciding complete equivalence is NP-complete as well. □

Nevertheless, we can still use the above DP formulation to construct a decision diagram in a top-down fashion as usual. It may not result a reduced decision diagram, but it correctly constructs a decision diagram representing the feasible set of the system of linear inequalities.

In this case, we can still check if each constraint is infeasible with respect to $\mathcal{D}$ or always satisfied by all solutions in $\mathcal{D}$, in the same manner as described in 2.7.1. If the former happens, the entire state is infeasible. If the latter happens, then within the state we replace the right-hand side of the associated constraint by a unique symbol such as $+\infty$, because any right-hand side corresponding to all feasible solutions in $\mathcal{D}$ can be treated as equivalent.

Another approach to construct a decision diagram for multiple linear inequalities is to construct one individually for each inequality and intersect them all. Intersecting decision diagrams is a well-known operation, described for instance in [12] in the context of linear inequalities and [58] in general. In a nutshell, its idea is to construct a new decision diagram with the following DP formulation. Let D_1 and D_2 be the two decision diagrams we are intersecting. Assume no long arcs for simplicity. Each state is associated to a pair of nodes (u_1, u_2) of the same layer, where u_1 is in D_1 and u_2 is in D_2. Given a value v and a state (u_1, u_2), the transition function returns the pair consisting of the endpoints of the v-arcs of u_1 and u_2 respectively if both are feasible. If at least one of them is infeasible – that is, it does not exist in the decision diagram – then the transition function returns the infeasible state $\hat{0}$. This operation can be performed for more than two decision diagrams simultaneously, in which case we replace the pairs by tuples of nodes in each decision diagram.

2.8 Decision Diagram Refinement

In order to construct good relaxed decision diagrams, we need them to have as few infeasible solutions as possible while keeping their sizes tractable. Decision diagram refinement techniques focus on improving a relaxed decision diagram in some way, typically removing infeasible solutions to make it a better approximation. It must also avoid substantially increasing the size of the diagram.

Refinement techniques include the following:

- *Arc filtering* consists of deleting arcs that only correspond to infeasible solutions. Its main advantage is that the relaxation is strengthened and at the same time the number of arcs of the decision diagram is reduced [6].

- *Node splitting* creates a copy of a chosen node, where outgoing arcs are duplicated and incoming arcs are partitioned between the two nodes. When the node and partition are properly chosen, node splitting enables further arc filtering – and thus elimination of infeasible solutions – at the cost of extra nodes [6, 37].

- *Constraint separation* eliminates all solutions that are infeasible with respect to a DP formulation. This is similar to constructing a decision diagram for the DP formulation and intersecting it with the original one, but this is done directly on the original structure [19].

From the techniques discussed above, we only further discuss arc filtering in this section – see the above references for more details on other techniques. Additionally, we introduce a natural way to adapt arc filtering to the compilation stage, which we call *arc pruning*, in Section 2.8.2. It uses bounds to remove infeasible arcs before exploring their endpoints.

2.8.1 Arc filtering

Arc filtering consists of identifying and removing arcs e such that all paths through e are infeasible with respect to a given constraint. In this section, we present filtering for the case of linear constraints, first discussed in [6].

Proposition 2.5. *Consider a decision diagram D with ordered variables $x_1, \dots, x_n$, an arc (u_1, u_2) of D assigning v to x_k, and a linear constraint $\sum_{i=1}^{n} a_i x_i \leq b$. Then no solution x corresponding to paths in D containing (u_1, u_2) satisfies the linear constraint if and only if*

$$\min_{(x_1,\dots,x_{k-1}) \in S^{\uparrow}(u_1)} \left\{ \sum_{i=1}^{k-1} a_i x_i \right\} + v + \min_{(x_{k+1},\dots,x_n) \in S^{\downarrow}(u_2)} \left\{ \sum_{i=k+1}^{n} a_i x_i \right\} > b$$

Proof. We can rewrite this condition as $\min_{x \in S_{(u_1,u_2)}} \{\sum_{i=1}^{n} a_i x_i\} > b$, where $S_{(u_1,u_2)}$ is the set of solutions corresponding to paths containing (u_1, u_2). This is equivalent to stating that all $x \in S_{(u_1,u_2)}$ violate the constraint. □

We call the two minimization values above the partial solution value of u_1 and the completion value of u_2 respectively. We can compute the partial solution values of all nodes with a single top-down pass, and similarly all completion values with one bottom-up pass. Therefore, arc filtering can be efficiently performed for all arcs at the cost of two passes through the decision diagram.

This approach can be naturally extended to constraints of the form $g(x) \leq b$ for any additively separable function $g(x)$, since the partial solution and completion values above can be efficiently computed as well.

2.8.2 Arc pruning

The arc filtering method described in the previous section is a post-processing approach. It may be useful however to perform arc filtering during compilation, even if approximately, in order to avoid unnecessarily visiting infeasible nodes. Arc pruning checks if a newly created arc is infeasible by following the same rules as arc filtering, except that it relies on bounds instead.

While arc pruning can be applied with respect to constraints not captured by the DP formulation, it can also be used with respect to a primal or a dual bound. In other words, given a primal bound P or a dual bound D and an objective c we are maximizing for example, we may prune arcs with respect to $c^\top x \geq P$ and $c^\top x \leq D$. We call these approaches *primal pruning* and *dual pruning* respectively.

We discuss the case of a linear constraint $a^\top x \leq b$ and assume it is not already implied by the DP formulation used to construct the decision diagram.

Top-down compilation

In a top-down compilation, the partial solution set of every node we explore is fully constructed. This allows us to compute the partial solution value in arc filtering in an exact manner. However, we replace the completion value by a lower bound (assuming a less-or-equal constraint), which we call *completion bound*. The following proposition is a straightforward extension of Proposition 2.5 which replaces an exact completion value by a completion bound.

Proposition 2.6. *Consider a decision diagram D ordered $x_1, \ldots, x_n$, an arc (u_1, u_2) of D assigning v to x_k, and a linear constraint $\sum_{i=1}^n a_i x_i \leq b$. Let B be a completion bound for u_2. That is, $B \leq \min_{(x_{k+1},\ldots,x_n)\in S^\downarrow(u_2)} \left\{\sum_{i=k+1}^n a_i x_i\right\}$. If $\min_{(x_1,\ldots,x_{k-1})\in S^\uparrow(u_1)} \left\{\sum_{i=1}^{k-1} a_i x_i\right\} + v + B > b$, then no solution x corresponding to paths in D containing (u_1, u_2) satisfies the linear constraint.*

Proof. This condition implies the condition in Proposition 2.5. □

In other words, given a completion bound B for a newly created arc (u_1, u_2), we can check the above condition to decide whether we can remove the arc or not. The partial value of each node can be recursively computed along with construction.

A question is how to compute the completion bound B for u_2. At this point, the only information we have at hand about the completion set of u_2 is its state. Thus, we compute B based on its state, which may be problem-specific.

Typically, a state contains information about a relaxation of the completion set. For instance, in cases like the maximum independent set problem, the state consists of domains of the completion variables. In this case, we can minimize $\sum_{i=k+1}^n a_i x_i$ over the domain given by the state. In other words, if $\mathcal{D}_s(x_i)$ corresponds to the domain of x_i for the state s, we let $B = \sum_{i:a_i\geq 0} \min(\mathcal{D}_s(x_i)) + \sum_{i:a_i<0} \max(\mathcal{D}_s(x_i))$.

For the case of multiple linear inequalities described in Section 2.7.2, a state represents linear inequalities for the completion set. Thus, we can compute B by minimizing $\sum_{i=k+1}^{n} a_i x_i$ over these linear inequalities. This is similar to the pruning mechanism in branch-and-bound trees in MIP solvers. However, this procedure may be expensive.

Note that arc filtering dominates arc pruning in terms of strengthening the relaxation, as we are performing the same operation as arc filtering except that we use a weaker bound. More precisely, the decision diagram obtained from arc filtering is a subgraph of the one from arc pruning, assuming a standard exact construction (e.g. no merge-based relaxations).

Depth-first compilation

Arc pruning can also be done within a depth-first construction, but it can be more expensive in terms of time. The main difference between the top-down and the depth-first cases is that the partial solution set $S^{\uparrow}(u)$ of a node u is not fully constructed in depth-first. Therefore, we may only have an upper bound B' for $\min_{(x_1,\dots,x_{k-1})\in S^{\uparrow}(u_1)} \left\{\sum_{i=1}^{k-1} a_i x_i\right\}$. If we compute the pruning condition using B' in place of the exact partial solution value, we may prune an arc that in fact should not be pruned because $S^{\uparrow}(u)$ is incomplete.

Nevertheless, we can postpone the exploration of the arc to whenever it stops satisfying the pruning condition (if ever), as follows. We keep the bound B' at each node corresponding to the partial solutions explored so far. Whenever the pruning condition is satisfied, we do not explore the new node, but also do not delete it. Throughout compilation, every time we merge equivalent nodes, we update its bound B' according to the new arc and propagate this update to its descendants, in the same top-down fashion used to compute partial solution values. During this update, we also allow previously "pruned" nodes to be updated. If the pruning condition no longer holds with the new bound, then we move it back to the list of open nodes, allowing it to be explored in a future iteration.

While this procedure may be relatively costly due to the constant partial solution value updating, it may help avoid unnecessarily visiting nodes.

Although the depth-first compilation limits the availability of the partial solution set, it provides us with full completion sets of some of the nodes. A natural question is whether we can leverage the completion sets in order to improve the completion bound or even avoid computing it from the state.

This can be done for the case of a single linear inequality, which has the special property that equivalence classes are ordered by inclusion of the completion set. Recall from Section 2.7.1 that the equivalence classes for a linear inequality are intervals $[L_i, U_i)$, which are nonoverlapping and ordered within a layer.

Suppose that we are deciding whether to prune an arc (u_1, u_2). For the sake of simplicity, assume we have the exact partial solution value of u_1, since the previously discussed updating procedure

eventually finds it. Moreover, assume that all nodes with fully constructed completion sets store their completion values. Let b be the state of u_2.

If u_2 is equivalent to any of the fully explored nodes, then we can use the equivalent node's completion value to prune the arc. Since we are using exact values, this has the same effect as filtering the arc as discussed in the previous section.

If u_2 is not equivalent to any of the nodes, then find the smallest L_i greater than b. Let u^* be the node corresponding to $[L_i, U_i)$. The fact that $b < L_i$ means that the inequality corresponding to b is tighter than any inequality associated to the interval (assuming less-or-equal inequalities). Thus, $S^{\downarrow}(u_2) \subseteq S^{\downarrow}(u^*)$. Therefore, the completion value B of u^* must be an upper bound for the completion value of u_2. Using B, check if the pruning condition is satisfied; if so, we can prune the arc (u_1, u_2).

2.8.3 Complexity of arc filtering

In Section 2.8.1, we have discussed how to identify whether all paths through a given arc are infeasible with respect to a constraint $a^\top x \leq b$. However, filtering arcs may be more difficult if we consider a different constraint. For instance, it may be the case that all paths through a given arc are infeasible with respect to the conjunction of linear constraints $Ax \leq b$, and yet not all of them are infeasible with respect to a single constraint in this set. We show in this section that it is NP-hard to identify such arcs.

Arc filtering is often discussed in the context of consistency. Establishing *DD consistency* consists of filtering every possible arc with respect to a given constraint, ensuring that every arc belongs to a feasible path. Arc filtering and DD consistency are polynomially equivalent in terms of complexity. Nevertheless, the proofs in this section are written in terms of arc filtering.

The complexity of arc filtering is closely connected to the complexity of deciding feasibility. Recall that $\mathcal{D}$ corresponds to the discrete domains of the n variables of the problem.

Proposition 2.7. *If deciding feasibility within $\mathcal{D}$ for a given constraint is NP-complete, then establishing DD consistency for this constraint is NP-hard.*

Proof. Consider the reduced decision diagram representing the set $\{0\} \times \mathcal{D}$, which has width 1 and $n + 2$ nodes. Let x_0 be the additional variable corresponding to the domain $\{0\}$. The problem of filtering the arc $x_0 = 0$ with respect to a given constraint is equivalent to deciding feasibility within $\mathcal{D}$ of that constraint.

□

The converse is not true: for instance, feasibility for the alldifferent constraint can be decided in polynomial time, but DD consistency for it is NP-hard [6].

In fact, DD consistency is more closely connected to a different feasibility problem. Define the *DD feasibility* problem for a problem or class of constraints as follows: given a decision diagram D

and an instance I of the problem or class of constraints, decide whether there exists any solution in D that is feasible with respect to I. A similar result to the proposition below is given in [39].

Proposition 2.8. *Deciding DD feasibility for a given constraint is NP-complete if and only if establishing DD consistency for this constraint is NP-hard.*

Proof. Given a constraint, filtering an arc is equivalent to deciding DD feasibility for the decision diagram consisting of the paths containing the arc. Therefore, if we can decide DD feasibility in polynomial time, we can filter arcs in polynomial time.

Conversely, suppose we can perform arc filtering in any decision diagram in polynomial time with respect to a constraint. Fix a decision diagram D. Similarly to the proof of Proposition 2.7, extend D by adding a new variable x_0 with domain $\{0\}$, not present in the constraint. If we can filter the new arc corresponding to $x_0 = 0$ in polynomial time, then we can decide DD feasibility with respect to D in polynomial time. □

We emphasize that DD feasibility takes a decision diagram as input. On the other hand, if the decision diagram is fixed a priori, then it is no longer true that feasibility is as hard as consistency. In particular, consider the case where the decision diagram is fixed to $\mathcal{D}$. There exist problems in which domain feasibility is easy but establishing domain consistency is NP-hard; see e.g. [53].

Moreover, establishing DD consistency is at least as hard as determining complete equivalence.

Proposition 2.9. *Let $\mathcal{P}$ be a problem or constraint set with discrete domains. Suppose an infeasible instance for $\mathcal{P}$ exists and is given. If it is NP-hard to establish DD consistency for $\mathcal{P}$, then it is NP-complete to decide complete equivalence for $\mathcal{P}$, whether in a top-down or a depth-first compilation.*

Proof. Suppose that we have a polynomial-time algorithm for determining complete equivalence with respect to $\mathcal{P}$. Then we can filter an arc by checking if its tail is equivalent to an infeasible state. In the depth-first case, we assume the infeasible state is the one with the completion set constructed. □

This also serves as an alternative proof for Proposition 2.1. By Proposition 2.7, hardness of feasibility implies hardness of arc filtering, which implies hardness of complete equivalence by Proposition 2.9.

A direct consequence of Proposition 2.7 is that establishing DD consistency with respect to two or more linear inequalities is NP-hard.

Corollary 2.10. *It is NP-hard to establish DD consistency with respect to a conjunction of two or more linear inequalities simultaneously in a decision diagram, even if it is binary.*

Proof. Deciding feasibility of a 0-1 knapsack problem with two or more linear inequalities is NP-complete. By Proposition 2.7, DD consistency is NP-hard. □

A slightly different question is whether we can find in polynomial time an inequality that is valid with respect to a polyhedron P in order to filter an arc. The answer to this question is positive. Nevertheless, the purpose of this result is mainly theoretical as the algorithm we provide as presented is impractical for establishing DD consistency, given that it entails solving a linear program for every arc of the decision diagram.

This can be viewed as the following problem: given the set of points S_e corresponding to arc e in a decision diagram, find a hyperplane that separates P from S_e or decide none exists.

Proposition 2.11. *Consider a decision diagram with an arc e and let $P = \{x \in \mathbb{R}^n : Ax \leq b\}$. There exists a polynomial-time algorithm to identify a valid inequality for P such that arc filtering can be applied with respect to e, or decide that no such inequality exists.*

Proof. Suppose that we want to filter arc e and let S_e be the set of solutions that use arc e. It suffices to show that we can find an inequality that separates P from $\text{conv}(S_e)$ in polynomial time.

A valid inequality with respect to P has the form $\lambda^\top Ax \leq \lambda^\top b$ for some $\lambda \geq 0$. Therefore, we can express the question as finding a $\lambda \geq 0$ such that this inequality is violated by all $x \in \text{conv}(S_e)$. In other words, $\lambda^\top(b - Ax) < 0$ for all $x \in \text{conv}(S_e)$. We can find λ by solving $\min_{\lambda \geq 0} \max_{x \in \text{conv}(S_e)} \lambda^\top(b - Ax)$. If the result is less than zero, then we have found an inequality that separates $\text{conv}(S_e)$ from P, or otherwise none exists. We show that this problem can be reformulated as a linear program.

In Chapter 4, we show that we can formulate the convex hull of S_e as a linear program. Theorem 4.1 states that $\text{conv}(S_e) = \text{proj}_x(P_{\text{flow}}(D_e))$, where D_e is the decision diagram corresponding to S_e and $P_{\text{flow}}(D_e)$ is as defined in the theorem. Therefore, this problem can be reformulated as $\min_{\lambda \geq 0} \max_{(x,f)} \{\lambda^\top b - \lambda^\top Ax : (x, f) \in P_{\text{flow}}\}$.

Since $\text{conv}(S_e)$ is nonempty, we can use strong duality in the inner problem. We obtain $\min_{\lambda \geq 0, u, v} \{\lambda^\top b + v_s : (\lambda, u, v) \in \hat{P}(D_e)\}$, where $\hat{P}(D_e) = \{(\lambda, u, v) : v_j \leq v_i - \ell_{ij} u_k \; \forall$ layer-k arc (i, j) in $D_e, v_t = 0, u + \lambda^\top A = 0\}$ and ℓ_{ij} corresponds to the label of arc (i, j).

This linear program can be solved in polynomial time and thus arc filtering can be done in polynomial time. □

2.9 Summary of Complexity of Equivalence and Consistency

Table 2.1 summarizes the complexity of complete equivalence and DD consistency for several problems and constraints. We describe each constraint class and provide references for the results below.

From the discussions in Sections 2.4 and 2.8.3, polynomial-time top-down complete equivalence implies polynomial-time depth-first complete equivalence, which in turn implies polynomial-time DD consistency. Equivalently, hardness results are implied from right to left in Table 2.1. In the descriptions below, we omit the complexity results that are consequences of these implications.

	Complete equivalence		
	Top-down	Depth-first	DD consistency
Set packing	P	P	P
Set covering	P	P	P
Set partitioning	NP-complete	NP-complete	NP-hard
Single linear inequality	NP-complete	P	P
Single additively separable inequality	NP-complete	P	P
Single linear equality	NP-complete	NP-complete	NP-hard
Multiple (2+) linear constraints	NP-complete	NP-complete	NP-hard
2-SAT	P	P	P
3-SAT	NP-complete	NP-complete	NP-hard
Alldifferent	NP-complete	NP-complete	NP-hard
Among	P	P	P
Element	P	P	P
Sequence	NP-complete	NP-complete	NP-hard

Table 2.1: Complexity results of node equivalence and DD consistency for classes of constraints. Domains are binary for set packing, set covering, set partitioning, conflict constraints, 2-SAT, and 3-SAT. We assume the remaining classes are under arbitrary bounded integer domains.

In addition, we use the following simple proposition to show polynomial-time complete equivalence for some of the constraints.

Proposition 2.12. *Let $\mathcal{P}$ be a problem or constraint set. If a decision diagram for $\mathcal{P}$ can be constructed in polynomial time and space, then complete equivalence for $\mathcal{P}$ can be performed in polynomial time and space, whether in a top-down or a depth-first compilation.*

Proof. Consider the following polynomial-time algorithm to perform an equivalence test given states s_1 and s_2. By the assumption, we can construct in polynomial time and space a decision diagram D for $\mathcal{P}$ and reduce it afterwards. Take any two partial solutions p_1 and p_2 leading to s_1 and s_2 respectively. Since D is reduced, the states are equivalent if and only if these partial solutions lead to the same node in D. □

Details of the complexity results are summarized below.

Set packing. Set packing constraints are constraints of the form $Ax \leq 1$, where A is a binary matrix and x is a binary vector of variables. A DP formulation that uses states as domains naturally defines complete equivalence [22].

Set covering. Set covering constraints are constraints of the form $Ax \geq 1$, where A is a binary matrix and x is a binary vector of variables. A DP formulation for set covering can be obtained by keeping track of whether a set covering constraint has been satisfied or not. Complete equivalence can be achieved if we always remove all redundant constraints from the state [17].

Set partitioning. Set partitioning constraints are constraints of the form $Ax = 1$, where A is a binary matrix and x is a binary vector of variables. In general, deciding feasibility of set partitioning constraints is NP-complete, and thus by Proposition 2.7, performing DD consistency is NP-hard.

Above, we assume multiple set partitioning constraints. If we consider a single set partitioning constraint (or more generally, a single cardinality constraint $\sum_{i=1}^{n} x_i = b$) equivalence and consistency are in P. Since the cardinality constraint has at most $n + 1$ states per layer, one for each possible value of left-hand side, a decision diagram can be constructed in polynomial time, and thus by Proposition 2.12, top-down complete equivalence is in P.

Single linear (or additively separable) inequality. By Proposition 2.2, top-down complete equivalence is NP-complete. Moreover, depth-first complete equivalence is in P as discussed in Section 2.7.1.

Single linear equality. Deciding integer feasibility of a single linear equality is NP-complete (even for binary domains), and thus DD consistency is NP-hard by Proposition 2.7. A pseudopolynomial time approach for DD consistency is described in Chapter 9.4 of [19], which consists of keeping track of all possible left-hand side values.

Multiple (2+) linear constraints. DD consistency is NP-hard as shown in Proposition 2.10.

2-SAT. 2-SAT is given by the conjunction of clauses, each of which is a disjunction of two literals, which may be variables or negated variables. 2-SAT constraints can be rewritten as implication constraints and we show in Section 3.3 that top-down complete equivalence for these constraints is in P.

3-SAT. 3-SAT is similar to 2-SAT, except that each clause is a disjunction of three literals. Deciding feasibility of 3-SAT is NP-complete, and thus DD consistency is NP-hard by Proposition 2.7.

Alldifferent. The alldifferent constraint requires that values taken by a set of variables are all distinct from each other. DD consistency is NP-hard for alldifferent; see Chapter 9.4.4 of [19], or [6].

Among. The among constraint requires that the number of variables taking values from a given set are between given lower and upper bounds. DD consistency is in P for among; see Chapter 9.4.5 of [19], or [39]. In addition, similarly to the cardinality constraint, the among constraint requires at most $n + 1$ states per layer and thus top-down complete equivalence is in P by Proposition 2.12.

A system of two or more among constraints generalizes set partitioning constraints, and thus DD consistency is NP-hard in this case.

Element. The element constraint is a binary constraint on variables x and y that enforces $y = c_x$ given $c_1, \ldots, c_m$, where $\mathcal{D}(x) = \{1, \ldots, m\}$ and $\mathcal{D}(y) = \{c_1, \ldots, c_m\}$. DD consistency is in P for element; see Chapter 9.4.6 of [19], or [39]. In addition, since the constraint is binary, we can construct its decision diagram efficiently and thus top-down complete equivalence is in P by Proposition 2.12.

Sequence. Given variables $x_1, \ldots, x_n$, the sequence constraint is a conjunction of an among constraint (with the same bounds and set of values) applied to all subsequences $x_i, \ldots, x_{i+q-1}$ of size q within the n variables. DD consistency is NP-hard for sequence; see Chapter 10 of [19], or [15].

Chapter 3

Decision Diagram Relaxations for Integer Programming Models

3.1 Introduction

A strength of decision diagrams lies in representing recursive structure embedded in certain discrete optimization problems. Typically, this structure needs to be explicitly modeled by a user through a dynamic programming formulation, which is often not readily available when facing a new problem. Instead, it is common for discrete optimization problems to be modeled via integer programming formulations, due to the effectiveness of mixed-integer programming solvers.

Providing both a DP and an IP formulation can improve the solving process of a MIP solver, as we will see for instance in Chapter 4. However, the user takes the burden of finding good formulations. Moreover, decision diagrams in their current form provide benefits only when this recursive structure provides substantial information beyond the IP model itself. Identifying such problems is challenging.

In this chapter, we discuss the following question: given an integer programming model, how to efficiently construct a decision diagram relaxation that approximates it well in practice? Approximation may have several different meanings and depends on the application. Here we focus on generating decision diagrams that yield strong dual bounds for the problem.

Dual bounds have several uses. In MIP solving, they typically come from LP relaxations and are employed to prune nodes in the branch-and-bound tree. They are also useful to guide the MIP search, such as deciding which variables to branch on with the strong branching technique, and to bound how far a primal feasible solution is from an optimal solution. In Chapter 5, we use them to improve pruning in the branch-and-bound tree of a MIP solver.

In the context of decision diagrams, good dual bounds can be obtained in practice for particular problems with specific DP formulations [22]. This work discusses how to generate dual bounds for more general problems modeled with integer programming.

This is a question that must be carefully approached. A first attempt to answer this question is to simply construct decision diagrams using the top-down compilation method discussed in Section 2.7.2. However, as we discuss in the computational section, this approach may lack the ability to take advantage of structure.

It is important to recognize that decision diagrams are tightly connected to the structure provided by DP formulations. Many real-world problems however are complex and do not adhere to an amenable structure. We cannot expect decision diagrams to aid an arbitrary integer programming model if we are unable to identify some structure within it. Nevertheless, they are often present as substructures. For instance, several problems contain set packing constraints – constraints of the form $Ax \leq 1$, where A is a binary matrix and x is a binary vector of variables – which, when isolated, tend to be receptive to decision diagram-based approaches [22].

Motivated by the diversity of real-world problems, we investigate how to generate dual bounds when a structure exploitable by decision diagrams is only partially present. We propose a framework that builds decision diagrams for classes of constraints present in the problem – which can be viewed as relaxations – and incorporates the remaining constraints via two approaches, Lagrangian relaxation and constraint propagation.

While this framework permits any choice of substructure, in this chapter we investigate the use of the conflict graph for binary problems. The conflict graph is a common component in modern MIP solvers and represents the pairs of binary variables that cannot both take a certain pair of values. It can be viewed as a relaxation of the problem and thus it fits our framework. Moreover, as we discuss in this chapter, the feasible set of a conflict graph admits a good DP formulation.

Although focusing on a particular substructure limits the range of applications, we do not aim to design a method to improve an arbitrary IP model. Instead, our approach is opportunistic: we only attempt to improve the solution process for a model when there are reasons to believe that decision diagrams can help – in this case, when conflict graphs are present and capture most of the problem. Future research may incorporate further classes of constraints and extend the reach of this framework. Furthermore, we emphasize that the approach does not require any input other than the IP model.

We begin by defining the framework in Section 3.2. Sections 3.3 and 3.4 detail two important aspects of the framework: constructing a reduced decision diagram for a conflict graph and handling constraints that are not considered in the decision diagram. Section 3.5 presents computational results.

3.2 Framework

A central challenge in designing approaches based on relaxed decision diagrams is to keep them small while still obtaining a good approximation of the problem. In a dual bound generation approach, this involves two main factors: the ability to identify equivalent nodes and the form of relaxation.

The power of decision diagrams comes from merging equivalent nodes. While in practice it is not vital that we merge every pair of equivalent nodes possible, merging as many as possible allows us to focus on other important factors that affect the size of decision diagrams. In other words, we want equivalence to be as close to complete as possible. Unfortunately, deciding complete equivalence for linear constraints is NP-complete (see Section 2.7), which motivates us to consider alternatives in this work. Moreover, even if we can attain complete equivalence, a decision diagram can still grow exponentially large. In order to manage its size, we must approximate the problem with a tractable relaxation.

In this framework, we consider two forms of relaxation: one at the level of decision diagram construction and another at the level of problem constraints.

At the decision diagram level, we construct relaxed decision diagrams using a top-down construction. We merge nodes in a way that heuristically avoids adding infeasible solutions of high objective value (when maximizing).

At the constraint level, the framework considers a substructure of the problem, such as a subset of constraints of specific type or, in the case of this work, conflict graphs. Not only substructures can be smaller, but more importantly information about problem structure can significantly benefit the construction of decision diagrams. However, this relaxation can be very weak if it ignores several constraints. This is compensated through the use of Lagrangian relaxation, which can be used with decision diagrams [16]. Moreover, we can partially incorporate them into the decision diagram through the use of constraint propagation.

Given that we use a substructure as the base for our decision diagram, we must to choose a structure with good qualities. We balance the following criteria in the choice of structure:

- **Identifiability:** We should be able to efficiently identify and extract the substructure from the problem. While this is trivial if we choose an explicit subset of constraints, we may also work with relaxations that are not explicitly given in the problem.

- **Generality:** The structure should be as generic as possible in order to capture structure within as many applications as possible. In particular, this structure must play a fundamental role in defining the problems we aim to improve upon, as otherwise the bounds generated would be weak.

- **Compactness:** In order to keep the size of the decision diagram compact, the formulation must ideally support efficient equivalence tests that are complete or close to being complete. Moreover, structures with good variable ordering and merging (relaxation) heuristics are desirable. It is well known that variable ordering can have a considerable effect on the size of the decision diagram [26] and likewise the quality of the bound from relaxed decision diagrams [23].

Conflict graphs satisfy these three criteria well. First, the task of identifying the conflict graph structure (when present) is already performed by modern MIP solvers, and thus we do not need to be concerned with extracting them. Second, the conflict graph encompasses common constraints such as set packing constraints and simple implications of the form $x_i = v_i \implies x_j = v_j$. In particular, it is equivalent to 2-SAT constraints (see e.g. [9] and note that conflict graphs are equivalent to implication graphs defined in the next section), and if we were to push it further to 3-SAT, complete equivalence becomes NP-hard as discussed in Section 2.9. Third, there exists a DP formulation for the conflict graph that has an efficient complete equivalence test in top-down construction, provided in Section 3.3. In addition, we generalize a variable ordering heuristic for the independent set problem, previously shown to perform well in practice [22].

We remark that the framework itself supports any type of structure, as long as we can efficiently build good decision diagrams from them. In this context, structure means any class of constraints that form a relaxation of the problem. For example, another choice is to focus on problems in which set covering constraints are present and construct decision diagrams from those constraints. In particular, combining multiple classes of constraints is a possibility that may enrich the generality of the method. However, we limit the scope of this work to the use of conflict graphs and leave other choices open as further research directions.

A summary of the framework is as follows.

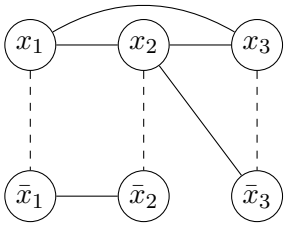
1. We select a substructure of the problem from which to construct a relaxed decision diagram – in this case, we use a conflict graph. Constraints that are not implied by the substructure are called *generic*.

2. We construct a decision diagram, possibly relaxed, using a DP formulation specific to the substructure (Section 3.3). During construction, we may propagate information from generic constraints into the decision diagram (Section 3.4.2).

3. Once the decision diagram is constructed, we apply Lagrangian relaxation in order to further incorporate generic constraints into the bound (Section 3.4.1).

3.3 Decision Diagrams for Conflict Graphs

The conflict graph captures constraints that forbid certain pairs of binary variables from taking specific values. More formally, a conflict graph $G = (V, E)$ is a graph with two vertices per binary variable of the problem. Each vertex corresponds to an assignment of 0 or 1 to the corresponding variable. Denote by x_j^v the node of the conflict graph corresponding to the assignment of v to the variable x_j. Following this notation, we use x_j^{1-v} to denote the node corresponding to the negation of x_j^v. An edge exists between x_i^u and x_j^v if the assignments $x_i = u$ and $x_j = v$ cannot simultaneously occur in a feasible solution of the problem. Figure 3.1 illustrates an example of a conflict graph.



$$x_1 + x_2 + x_3 \leq 1$$
$$x_2 + (1 - x_3) \leq 1$$
$$(1 - x_1) + (1 - x_2) \leq 1$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

Figure 3.1: Example of a conflict graph for three binary variables, where x_i and $\bar{x}_i$ indicate setting x_i to 1 and 0 respectively. On the right, a linear representation of the constraints from the conflict graph.

Conflict constraints can be inferred in MIP solvers when applying, for instance, bound strengthening or probing during a presolve step. A common use of a conflict graph is to generate cuts [10, 2].

Note that each conflict constraint on x_i^u and x_j^v is equivalent to the constraint $x_i = u \implies x_j = 1 - v$, which is itself equivalent to $x_j = v \implies x_i = 1 - u$. Therefore, we can express conflict constraints as implications by replacing each edge $\{x_i^u, x_j^v\}$ with a pair of directed arcs (x_i^u, x_j^{1-v}) and (x_j^v, x_i^{1-u}). The resulting graph is called an implication graph. Since this conversion can occur in both directions, conflict graphs are equivalent to implication graphs.

Throughout this section, it is more convenient to describe a formulation for the implication graph instead of the conflict graph. Concepts from this formulation can be directly translated to the context of the conflict graph through the above equivalence.

We remark that modern MIP solvers may construct implication graphs for general integer variables instead of binary [2]. However, in this work, we focus on the binary setting.

3.3.1 Dynamic programming formulation

We provide a dynamic programming formulation for the feasible set of the implication graph – that is, the set of all solutions that satisfy the implication constraints encoded in the graph.

For notational convenience, we assume that variables are ordered as $x_1, \ldots, x_n$. The layer j (or stage j in DP terms) contains the states in which we have defined assignments for variables $x_1, \ldots, x_{j-1}$ and seek to assign values to $x_j, \ldots, x_n$. These values are restricted to the variable's domain, denoted by $\mathcal{D}(x_j)$ for each variable x_j, which in this chapter we assume to be $\{0, 1\}$. We denote the infeasible state by $\hat{0}$.

Each state at layer j is represented by domains of the variables $x_j, \ldots, x_n$. When we transition by setting the variable x_j to v_j, we remove from the domains all assignments $x_k = v_k$ such that $x_k^{1-v_k}$ is reachable from $x_j^{v_j}$. Here, we say that u is reachable from v if there exists a directed path in the implication graph from u to v. In other words, we take the implied assignments and remove their complements from the domains.

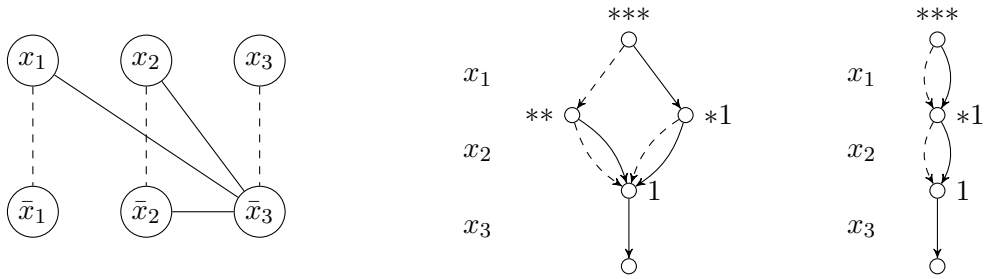


Figure 3.2: On the left, a conflict graph, where x_i and $\bar{x}_i$ indicate setting x_i to 1 and 0 respectively. Next, the decision diagram that would be obtained by using the DP formulation (IG) as is. The states are depicted as a sequence of symbols representing the domain of each completion variable in order: $*$ if the domain is $\{0,1\}$, 1 if it is $\{1\}$, and 0 if it is $\{0\}$. On the right, the decision diagram obtained if we establish domain consistency at the root state, which is always reduced as proved in Theorem 3.3.

More precisely, the DP formulation (IG) is defined as follows:

State space. For each stage j, each state $s \in \mathcal{S}^j$ is a list $(\mathcal{D}(x_j), \mathcal{D}(x_{j+1}), \dots, \mathcal{D}(x_n))$ such that each $\mathcal{D}(x_j)$ represents the domain of the variable x_j, and thus may take the values $\varnothing$, $\{0\}$, $\{1\}$, or $\{0,1\}$ in this binary setting.

Transition function. Let $\mathcal{D}_s$ be the domain associated to state s. Given an assignment v_j to a variable x_j, denote by $\mathcal{D}'_s(x_k) = \mathcal{D}_s(x_k) \setminus \bar{R}_{k,j,v_j}$, where $\bar{R}_{k,j,v_j}$ is the set of values v_k such that the node $x_k^{1-v_k}$ is reachable from $x_j^{v_j}$ in the implication graph G. The transition function t_j at layer j is defined as:

$$t_j(s, v_j) = \begin{cases} (\mathcal{D}'_s(x_{j+1}), \mathcal{D}'_s(x_{j+2}), \dots, \mathcal{D}'_s(x_n)) & \text{if } v_j \in \mathcal{D}_s(x_j) \text{ and } \mathcal{D}'_s(x_k) \neq \varnothing \\ & \qquad \text{for all } k = j+1, \dots, n, \\ \hat{0} & \text{otherwise.} \end{cases}$$

This DP formulation can be used to construct a decision diagram as described in Chapter 2 using its natural equivalence test: two nodes are equivalent if the states are the same. Figure 3.2 illustrates a decision diagram constructed from this DP formulation.

Optionally, the depth of the search tree from $x_j^{v_j}$ used to find reachable nodes may be limited for purposes of efficiency. This may affect the completeness of equivalence, but its correctness only requires us to update the domains of the neighbors of $x_j^{v_j}$.

We next show that this formulation is correct and provide a sufficient condition for complete equivalence that can be efficiently guaranteed.

Correctness

The proposition below shows that (IG) models the implication graph.

Proposition 3.1. *The DP formulation (IG) is correctly models the feasible set of the given implication graph G.*

Proof. Let D be the decision diagram generated by (IG), $\text{Sol}(D)$ be the set of solutions represented by s-t paths in D, and $\text{Sol}(G)$ be the feasible set of the implication graph G. We want to show that $\text{Sol}(D) = \text{Sol}(G)$.

The implication constraints of G enforce that if x_j is set to v_j, then x_k must be set to v_k for all nodes $x_k^{1-v_k}$ that are reachable from $x_j^{v_j}$. Since the transition function only enforces these constraints, it cannot eliminate feasible solutions. This implies that all feasible solutions must be represented as s-t paths in D. Therefore, $\text{Sol}(G) \subseteq \text{Sol}(D)$.

To show that $\text{Sol}(D) \subseteq \text{Sol}(G)$, let $\hat{x}$ be a solution represented by an s-t path in D. We want to show that $\hat{x}$ is feasible with respect to G. Suppose for contradiction that $\hat{x}$ is infeasible. Then $\hat{x}$ must violate the constraint of some arc $(x_j^{v_j}, x_k^{v_k})$ in G. That is, $\hat{x}_j = v_j$ and $\hat{x}_k = 1 - v_k$. Assume without loss of generality that x_j comes before x_k in the ordering, which can be done because each arc $(x_j^{v_j}, x_k^{v_k})$ in G has a counterpart $(x_k^{1-v_k}, x_j^{1-v_j})$. Then this assignment cannot occur because $t_j(s^j, v_j)$ enforces the domain of x_k to become $\{v_k\}$ in all subsequent states, which is a contradiction. Therefore, $\hat{x}$ is feasible, and thus $\text{Sol}(D) = \text{Sol}(G)$. □

In fact, formulation (IG) is correct even in a depth-d variant for any $d \geq 1$, in which we redefine $\bar{R}_{k,j,v_j}$ in the transition function to only consider nodes within a distance of d from $x_j^{v_j}$. Note that the exact same proof above holds in this case.

Completeness

Now that the correctness of (IG) is established, we turn to the question of when this formulation yields a reduced decision diagram.

The example in Figure 3.2 shows that (IG) as currently formulated does not always generate a reduced decision diagram. In the first decision diagram depicted on Figure 3.2, the two second-layer nodes have the same completion set but are not merged together. The state s of the node on the left unnecessarily has 0 in $\mathcal{D}_s(x_3)$, which if removed, would enable merging. This observation suggests the lemma below, which provides a sufficient condition for completeness.

We call a state s domain consistent if for every variable x_i and value $v_i \in \mathcal{D}_s(x_i)$, there exists a feasible completion from state s that assigns v_i to x_i.

Lemma 3.2. *If every state in (IG) is domain consistent, then the natural equivalence from formulation (IG) is complete.*

Proof. Domain consistency ensures that if two domains are different, then they must have different completion sets. More formally, consider states s_1 and s_2 with different domains. Suppose without loss of generality that there exists $v_i \in \mathcal{D}_{s_1}(x_i)$ such that $v_i \notin \mathcal{D}_{s_2}(x_i)$. Then the above property implies there exists a completion from s_1 which assigns v_i to x_i that does not exist from s_2. □

As a side note, observe that Lemma 3.2 holds not only for the formulation (IG), but also for any formulation in which the state space consists of domains, respected by their completions.

The next step is to provide a means to obtain domain consistency at every state, since this would yield completeness. In fact, it turns out that the transition function $t_j(s^j, v_j)$ in (IG) preserves domain consistency as long as the original state s^j is also domain consistent, as we establish in Theorem 3.3. This implies that it is sufficient to make the root state domain consistent.

For instance, in Figure 3.2 it would suffice to make the initial state domain consistent in order to construct a reduced decision diagram.

Theorem 3.3. *If s is a domain consistent state, then $t_j(s, v_j)$ is a domain consistent state if feasible.*

In order to prove Theorem 3.3, we first derive two intermediate lemmas. We use the following theorem from Aspvall et al. [9], which characterizes feasibility of an implication graph.

Theorem 3.4 (Aspvall et al. [9])**.** *An implication graph G is feasible if and only if there is no x_j such that x_j^0 and x_j^1 are in the same strongly connected component.*

The two intermediate lemmas are the following.

Lemma 3.5. *Given an implication graph G, there exists a feasible solution with x_j set to v_j if and only if there exists a feasible solution for the implication graph $\hat{G} := G \cup \{(x_j^{1-v_j}, x_j^{v_j})\}$.*

Proof. The constraint from the additional arc $(x_j^{1-v_j}, x_j^{v_j})$ is violated exclusively by all solutions with x_j set to $1 - v_j$, leaving exactly the feasible solutions of G with x_j set to v_j. □

Lemma 3.6. *Given a feasible implication graph G, there exists a feasible solution with x_j set to v_j if and only if there is no path from $x_j^{v_j}$ to $x_j^{1-v_j}$ in G.*

Proof. By Lemma 3.5, there is a feasible solution with x_j set to v_j if and only if $\hat{G} := G \cup \{(x_j^{1-v_j}, x_j^{v_j})\}$ is feasible. In view of Theorem 3.4 and the feasibility of G, $\hat{G}$ is feasible if and only if adding $(x_j^{1-v_j}, x_j^{v_j})$ to G keeps $x_j^{1-v_j}$ and $x_j^{v_j}$ in different strongly connected components.[1] This happens if and only if there is no path from $x_j^{v_j}$ to $x_j^{1-v_j}$ in G. □

[1] As a technicality, this requires that Theorem 3.4 holds when there are arcs between two nodes of the same variable. Despite assuming a standard implication graph, the proof from Aspvall et al. [9] is also valid with same-variable arcs.

Lemma 3.6 tells us how to achieve domain consistency for the implication graph. For every variable x_j, we check if x_j^0 is reachable from x_j^1 and vice versa. If $x_j^{1-v_j}$ is reachable from $x_j^{v_j}$, we remove v_j from $\mathcal{D}_s(x_j)$.

In addition, this can be done in linear time as follows. Tarjan's strongly connected components algorithm [55] provides the strongly connected components in reverse topological order. By treating each component as a node, we can scan the graph in topological order in a single pass to find these paths. Throughout this pass, we store at each component the variable-value assignments of its ancestors in order to pass it forward. Whenever we find the complement of one of these assignments, we can remove the assignment from the domain.

We now prove Theorem 3.3, which implies that it suffices to ensure domain consistency at the root state in order to guarantee completeness.

Proof of Theorem 3.3. Consider the states s and $s' := t_j(s, v_j)$. In order to show that s' is domain consistent, we need to show that for any $v_k \in \mathcal{D}_{s'}(x_k)$, there exists a completion from s' that assigns v_k to x_k.

Define $\hat{G}_s$ to be the implication graph G with the additional arcs $(x_j^{v_j}, x_j^{1-v_j})$ for all $v_j \in \{0,1\} \setminus \mathcal{D}_s(x_j)$. Note that the feasible set of $\hat{G}_s$ corresponds to the completion set of s by Lemma 3.5. Moreover, the feasible set of $G' := \hat{G}_s \cup \{(x_j^{1-v_j}, x_j^{v_j})\}$ corresponds to the completion set of s'. Following Lemma 3.6, it suffices to show that G' does not contain a path from $x_k^{v_k}$ to $x_k^{1-v_k}$.

Given that $\mathcal{D}_s$ is consistent, there must exist a completion x from s such that $x_k = v_k$. Equivalently, $\hat{G}_s$ must not contain a path from $x_k^{v_k}$ to $x_k^{1-v_k}$ according to Lemma 3.6. Therefore, any path in G' from $x_k^{v_k}$ to $x_k^{1-v_k}$ must go through the only new arc $(x_j^{1-v_j}, x_j^{v_j})$. However, $x_k^{1-v_k}$ is not reachable from $x_j^{v_j}$, as otherwise v_k would be removed from $\mathcal{D}_{s'}(x_k)$ as a result of the transition function. Hence, there cannot be a path in G' from $x_k^{v_k}$ to $x_k^{1-v_k}$. □

The above theorem directly implies the following result.

Corollary 3.7. *The natural equivalence from the DP formulation (IG) is complete when the initial state is domain consistent.*

Therefore, once we establish domain consistency in the root state, we can use the DP formulation in a top-down fashion to construct a reduced decision diagram.

We remark that this serves as an alternative proof for complete equivalence for the independent set problem in [22]. If we view the independent set problem in terms of an implication graph, we obtain a graph where all arcs point from a nonnegated node to a negated node. This means that every path in the implication graph has length at most one, and thus it suffices for the transition function to consider only the neighbors of each vertex, as done in the formulation. Moreover, the initial domain of all possibilities is always consistent since every individual vertex of the original graph is a feasible independent set.

3.3.2 Variable ordering

Variable ordering for decision diagrams are often based on heuristics. Using a fast heuristic is particularly helpful in our case, as we may be generating several decision diagrams during the solution process of a single problem.

Based on its close connection to the independent set problem, we use a generalization of a variable ordering heuristic for independent set that has shown to work well in practice, namely the minimum number of states ordering [23, 22]. In the context of independent set, at each layer, the ordering selects the vertex v that appears in the fewest number of states in the state pool. Every node with a state in which v appears will branch to both zero and one, whereas if v does not appear, the corresponding node only branches to zero. Therefore, this minimizes the number of arcs in the following layer.

A natural generalization is as follows: at each layer, we select the variable with the smallest sum of domain sizes throughout the state pool. This minimizes the number of arcs in the next layer since each assignment corresponds to an arc, given that the domains are consistent.

3.4 Generic Constraints

Focusing on substructures is typically only practical if the generic constraints are still taken into account in some form. We generally assume that the substructure consists of a significant part of the problem and these generic constraints are not worth completely including into the relaxed decision diagram.

In the context of conflict graphs, we mark a constraint as generic in our implementation if it does not have a particular form implied by conflict constraints: $\sum_{i \in P} x_i + \sum_{i \in N} (1 - x_i) \leq 1$ for some disjoint set of variable indices P and N. Although this can be checked quickly, it is possible that we label as generic more constraints than necessary.

For an arbitrary substructure, identifying generic constraints can be done by checking if every solution represented in the decision diagram is satisfied by the constraint. In other words, a constraint $a^\top x \leq b$ can be marked as generic if $\max_{x \in S}\{a^\top x\} \leq b$ where S is the set of points represented by the decision diagram, which can be efficiently checked by finding a maximum weight path in the decision diagram.

We handle generic constraints into two ways: Lagrangian relaxation and constraint propagation.

3.4.1 Lagrangian relaxation

Lagrangian relaxation consists of moving a set of constraints to the objective function by penalizing its violation. More precisely, in our context we solve the following problem:

$$\min_{\lambda \geq 0} \max_{x} \{c^\top x + \lambda^\top (b - Ax) : x \in \text{conv}(S)\}$$

where S is the set of points represented by a relaxed decision diagram, $Ax \leq b$ are the generic constraints not incorporated into S, and c is the objective function of the problem. The variables λ are called Lagrange multipliers, which represent penalties for the violation of the constraints $Ax \leq b$.

This problem can be solved with subgradient methods that require optimizing a linear function over $\text{conv}(S)$ as a subproblem. In our context, this subproblem entails finding an optimal path in the decision diagram representing S, which can be efficiently done.

Lagrangian relaxation theory establishes that the solution of the above problem is equivalent to the solution of the following one:

$$\max_x \{c^\top x : Ax \leq b, x \in \text{conv}(S)\}. \tag{3.1}$$

This provides a clean interpretation of the bound we obtain from Lagrangian relaxation. Essentially, we are optimizing over the convex hull of the set of points represented by the decision diagram intersected with the generic constraints in their original linear form. In other words, we are convexifying the constraints involved in the construction of the decision diagram, taking integrality into account.

A limitation of Lagrangian relaxation is that it is only equivalent to adding the constraints back in its original linear form. In some cases, we may need to tighten these generic constraints in order to obtain improvements. For instance, if the decision diagram is constructed from a set of linear constraints whose polyhedron has only integer vertices, then this approach cannot yield a better bound than the LP bound.

3.4.2 Constraint propagation

Even if a generic linear constraint results in a large decision diagram by itself, it can be partially incorporated into the decision diagram of other constraints without significantly increasing its size. This is particularly true if we use domain states: we use constraint propagation to filter out infeasible values from the domain states [6, 39]. This results in the elimination of infeasible points from the decision diagram, which may improve the bounds generated. Moreover, it may reduce the time it takes to construct the decision diagram, as we are potentially exploring fewer nodes.

Consider a constraint $a^\top x \leq b$ and a node u with domain state s. Given a variable x_j and a value v_j in the domain $\mathcal{D}_s(x_j)$, our goal is to determine before branching on u if no completion assigning v_j to x_j satisfies the constraint. If so, we can remove v_j from the domain $\mathcal{D}_s(x_j)$.

Suppose for now that we want to tackle this problem on a fully constructed decision diagram. This is equivalent to determining if the constraint is violated by all possible solutions with $x_j = v_j$ corresponding to paths that pass through the node u. To solve this, we can find the smallest right-hand side $a^\top x$ within this solution set and check if it exceeds the right-hand side b. This can be expressed in terms of the partial solution set and the completion set of the node u as follows.

Recall that $S^{\uparrow}(u)$ and $S^{\downarrow}(u)$ denote the partial solution set and the completion set of node u respectively.

Proposition 3.8. *Consider a decision diagram D ordered $x_1, \dots, x_n$, a node u of D at layer k, and a linear constraint $\sum_{i=1}^{n} a_i x_i \leq b$. Let $j \geq k$. Then no solution x with $x_j = v_j$ corresponding to paths in D containing u satisfies the linear constraint if and only if*

$$\min_{(x_1,\dots,x_{k-1}) \in S^{\uparrow}(u)} \left\{ \sum_{i=1}^{k-1} a_i x_i \right\} + \min_{\substack{(x_k,\dots,x_n) \in S^{\downarrow}(u) \\ x_j = v_j}} \left\{ \sum_{i=k}^{n} a_i x_i \right\} > b$$

Proof. The set of solutions x corresponding to paths containing u is $S(u) := \{x : (x_1, \dots, x_{k-1}) \in S^{\uparrow}(u), (x_k, \dots, x_n) \in S^{\downarrow}(u)\}$. Let $S'(u) := S(u) \cap \{x : x_j = v_j\}$, which is the set of solutions for which we want to check violation. This set violates the constraint if and only if $\min_{x \in S'(u)} \sum_{i=1}^{n} a_i x_i > b$, which is equivalent to the above condition. □

Denote the two terms in the condition above by $p_a(u) := \min_{(x_1,\dots,x_{k-1}) \in S^{\uparrow}(u)} \{\sum_{i=1}^{k-1} a_i x_i\}$ and $c_a(u, x_j, v_j) := \min_{(x_k,\dots,x_n) \in S^{\downarrow}(u), x_j = v_j} \{\sum_{i=k}^{n} a_i x_i\}$ respectively.

We return to the context of filtering the domain of a node u at the construction stage. We can efficiently compute $p_a(u)$, since it consists of optimizing a linear function over the decision diagram of the partial solution set of u, which is fully available at the time of branching for a top-down construction. It is not necessary to recompute $p_a(u)$ at every node, as we can maintain them throughout the construction. At every new node u' coming from a node u and arc $x_j = v_j$, we let $p_a(u') := p_a(u) + a_j v_j$. In addition, whenever two nodes u and u' are merged into u'', we let $p_a(u'') = \min\{p_a(u), p_a(u')\}$.

On the other hand, computing $c_a(u, x_j, v_j)$ during construction is difficult because we do not have the completion set of the node. Instead, we compute a lower bound B for $c_a(u, x_j, v_j)$. If we satisfy the condition $p_a(u) + B > b$, then by Proposition 3.8 we can still safely remove v_j from $\mathcal{D}_s(x_j)$. In our case where domains are states, we calculate B by minimizing $\sum_{i=k}^{n} a_i x_i$ over the possible values of the domains, after restricting x_j to be v_j. More precisely, we let B be $\sum_{i:a_i \geq 0, i \neq j} \min(\mathcal{D}_s(x_i)) + \sum_{i:a_i < 0, i \neq j} \max(\mathcal{D}_s(x_i)) + a_j v_j$.

This completes the description of the constraint propagation method. We remark that while this approach can only improve the bound since it removes infeasible solutions, it can potentially increase the size of the decision diagram. A simple example where this happens is given in Figure 3.3.

Alternatively, if we want to ensure that the size of the decision diagram does not increase, we may apply propagation only with respect to the variable we are currently assigning. This is equivalent to arc pruning as described in Section 2.8.2.

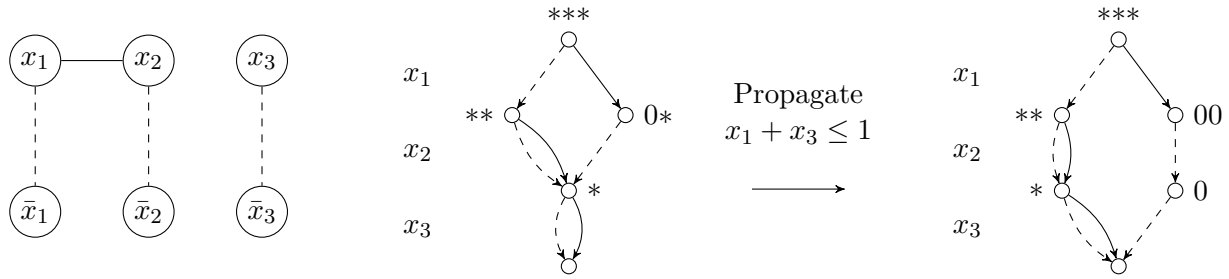


Figure 3.3: An example in which propagation increases the size of the decision diagram in the context of conflict graphs.

3.5 Computational Experiments

We computationally evaluate the dual bounds obtained from an implementation of the framework using conflict graphs. We limit these experiments to generating bounds at the root; in Chapter 5, we investigate the integration of these bounds into the branch-and-bound search process.

Along with the framework, we evaluate bounds obtained from a standard top-down construction based on linear constraints. We evaluate two versions: one in which the states are right-hand sides of constraints (as defined in Section 2.7.2) and one in which we augment these states with domains, which are used to restrict the possible transitions. In both cases, we check if each constraint is infeasible or will be always feasible with respect to the domain, as described in Section 2.7.2.

In the latter version, the domains are filtered via bounds propagation with respect to each constraint. After every transition, we remove from the domains any variable-value assignment such that every completion in the domain with this assignment violates a particular constraint. More precisely, at a state s with domains $\mathcal{D}_s$, for every constraint $a^\top x \leq b$, we remove from $\mathcal{D}_s$ every assignment $x_j = v_j$ for $v_j \in \mathcal{D}_s(x_j)$ such that $\min_{x \in \mathcal{D}_s, x_j = v_j}\{a^\top x\} > b$. Note that this construction yields a reduced decision diagram for set packing constraints, since it emulates the transition function for the independent set problem [22].

3.5.1 Experimental setup

As previously discussed, this approach is opportunistic in the sense that it only takes advantage of instances that contain conflict graphs that capture many of the constraints of the problem. Therefore, it makes sense to run experiments on instances with this substructure.

We randomly generate instances that are a combination of independent set (set packing) constraints – which provide the conflict graph structure – and knapsack constraints. The integer

programming model is the following:

$$\begin{aligned}
\max\ & c^\top x \\
& \sum_{j \in C} x_j \leq 1 \qquad \text{for all } C \in \mathcal{C} && \text{(set packing)} \\
& \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for all } i = 1, \ldots, m_{\text{knap}} && \text{(knapsack)} \\
& x \in \{0,1\}^n
\end{aligned}$$

The set $\mathcal{C}$ is a set of cliques that cover an input graph G – therefore, the above constraints effectively model independent set constraints for G. These cliques are derived in a greedy fashion: for each clique, we add vertices in nonincreasing order of degree (adjacent to all vertices in the current set) and remove them from the graph before the next clique is generated. The graph from which the independent set constraints are derived is a random graph following the Watts-Strogatz model [57], which has small-world properties and tend to work well with decision diagrams. It is generated as follows: given the desired number of vertices n, the desired mean degree k (assumed even), and a probability p, construct a preliminary graph with n vertices arranged in a cycle and two vertices are adjacent if and only if they are within distance $k/2$ in the cycle. Then for each vertex i and outgoing edge (i, j), reassign j with probability p to another vertex (besides i or a neighbor of i) uniformly chosen at random.

The m_{knap} knapsack constraints have coefficients a_{ij} chosen uniformly at random from 1 to 100 with a support of constant size 100. That is, we select 100 variables uniformly at random and let the coefficients of the remaining variables be zero. We maximize an objective with coefficients c_j also randomly chosen from 1 to 100. In some of the experiments, we vary the number of knapsack constraints m_{knap} and the right-hand sides b_i. In those that we do not, they are fixed to $0.1n$ and 150 respectively.

We consider instances of sizes $n = 200$ and $n = 1000$, each of which 10 are generated. We use SCIP 5.0.1 as the MIP solver along with CPLEX 12.6 as the LP solver. In all runs, presolve is on, except for variable aggregation and restarts, which complicate the use of decision diagrams. Cutting planes are at their default settings. The experiments were performed on a 2.33Ghz Linux machine with 32GB of RAM.

The variable ordering we use for the linear constraint-based decision diagrams is the ordering given by the Cuthill-McKee heuristic, which attempts to minimize the bandwidth of a matrix – in this case, the matrix of coefficients. For the conflict graph, we use the ordering described in Section 3.3.2. We fix the relaxed decision diagram width to be 1000.

We evaluate a dual bound D in terms of gap with respect to a primal bound P, computed as $(D - P)/P$. The primal bound is the best bound after 10 minutes of solving with SCIP. It is exact for instances of size $n = 200$, but may not be for the other instances.

As a baseline, we include in the plots the initial LP bound and the LP bound at the end of the root node. This latter bound is stronger than the initial LP bound mainly due to cutting planes, but may also include the influence of heuristics and a first round of strong branching at the root.

In order to make the plots clearer, we plot bounds from linear constraints and from the conflict graph separately, since their scales may be widely different. The LP bounds, present in both sets of plots, can be used as a baseline to compare the approaches.

We vary three parameters of the instance class: number of variables n, number of knapsack constraints m_{knap}, and right-hand side b_i. The first allows us to see how the bound scales and the other two enables us to view the effect of the generic constraints on the bound.

Varying number of variables n. Figure 3.4 shows the bounds as we scale up the number of variables. A first observation is that the linear constraint-based decision diagrams scale poorly. If we take advantage of the conflict graph, the bounds scale significantly better. Nevertheless, in both cases, the bounds are stronger for small instances.

This behavior suggests that relaxed decision diagrams may be appropriate for decomposition-type approaches. In fact, in Chapter 5, we apply the bounds on small nodes in a branch-and-bound tree.

Lagrangian relaxation and constraint propagation can be helpful especially for smaller instances. Their effect is more evident in the next set of plots.

Varying number of knapsack constraints m_{knap}. Figures 3.5 and 3.6 illustrate the change in the number of knapsack constraints for the cases of $n = 200$ and $n = 1000$ respectively. We can see that for small instances, the bounds are fairly strong even for several knapsack constraints. For the large instances, as expected more knapsack constraints make it harder to exploit the conflict graph. In particular, Lagrangian relaxation and constraint propagation are more helpful when more knapsack constraints are present.

On the other hand, bounds coming directly from linear constraints do not become worse as we add knapsack constraints – interestingly, they become better. This may indicate they are more suited for more general constraints.

Varying right-hand side b_i. In Figures 3.7 and 3.8, we vary the right-hand side in the knapsack constraint for the cases of $n = 200$ and $n = 1000$ respectively. The right-hand side serves as a lever to control the importance of the conflict graph. If the right-hand side is small, then the knapsack constraints dominate the problem. If the right-hand side is large, then the knapsack constraints become more redundant.

We observe especially in the plot for $n = 1000$ that the looser the right-hand side, the better are the bounds from the conflict graph. This is not evident from the linear constraint-based

construction. This supports the natural intuition that relaxed decision diagrams from a substructure perform better if that substructure plays a large role in defining the problem.

Note also that the Lagrangian relaxation ensures that the bound is worse than the LP bound when the right-hand side is small – though it may not do so in general given that we solve the Lagrangian relaxation approximately.

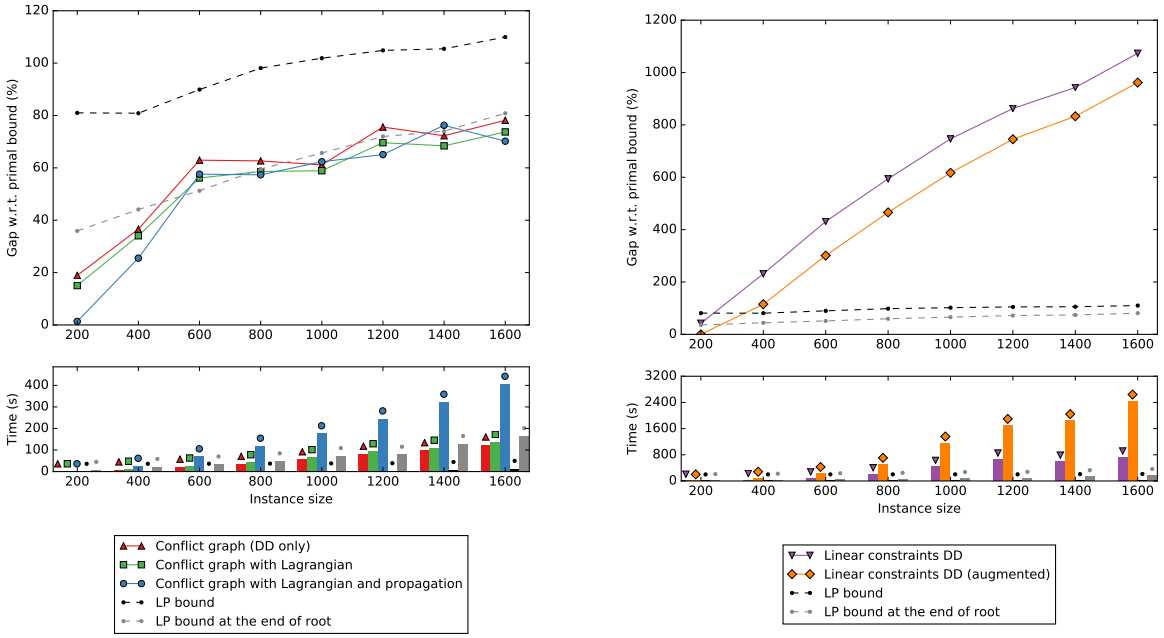


Figure 3.4: Dual bounds from relaxed decision diagrams for independent set + knapsack constraints, varying the number of variables.

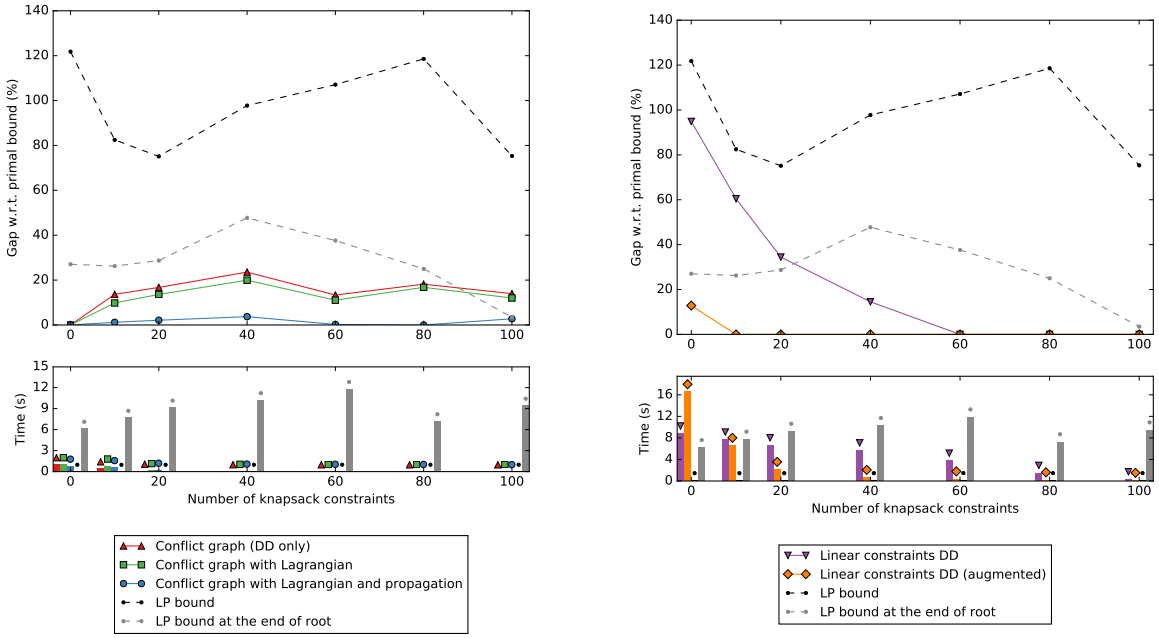


Figure 3.5: Dual bounds from relaxed decision diagrams for independent set + knapsack constraints with 200 variables, varying the number of knapsack constraints.

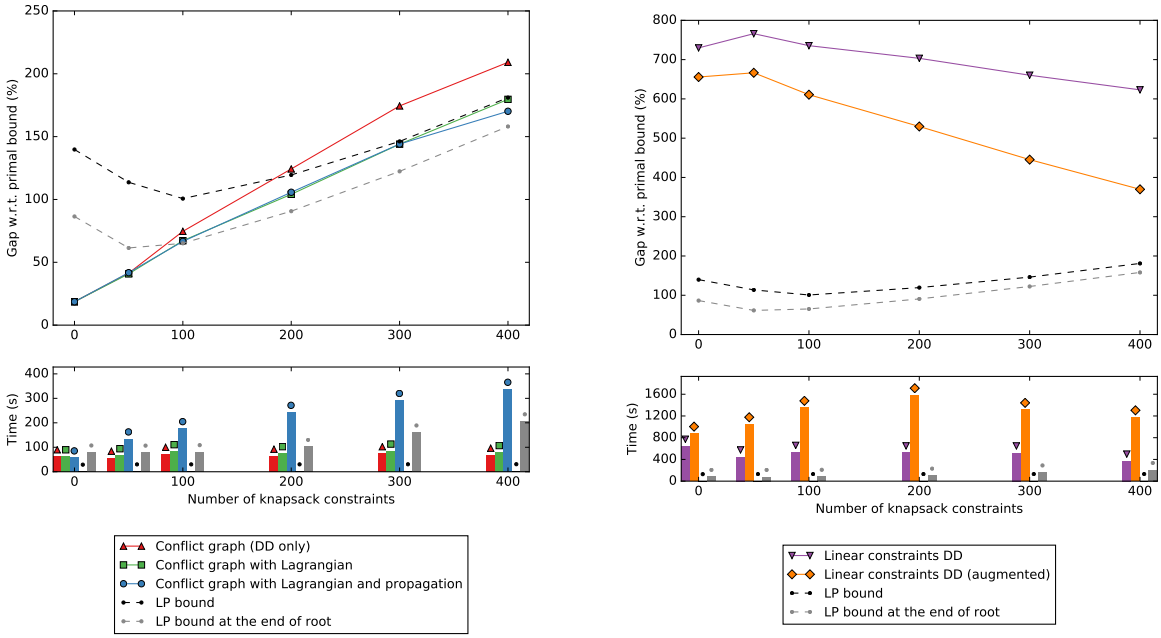


Figure 3.6: Dual bounds from relaxed decision diagrams for independent set + knapsack constraints with 1000 variables, varying the number of knapsack constraints.

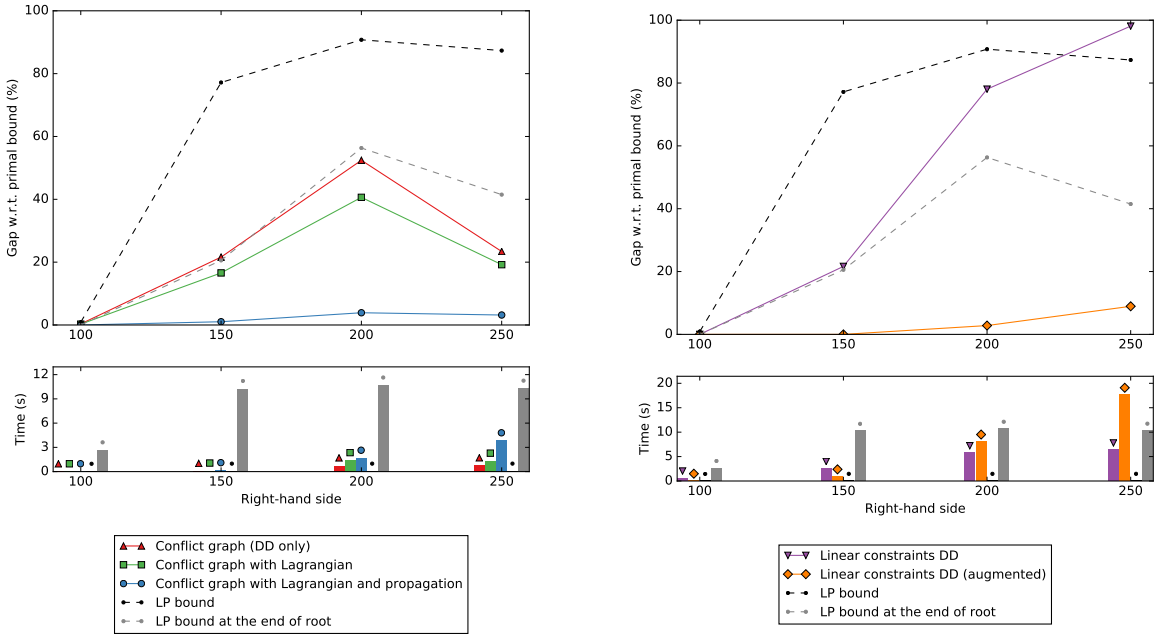


Figure 3.7: Dual bounds from relaxed decision diagrams for independent set + knapsack constraints with 200 variables, varying the right-hand side of the knapsack constraints.

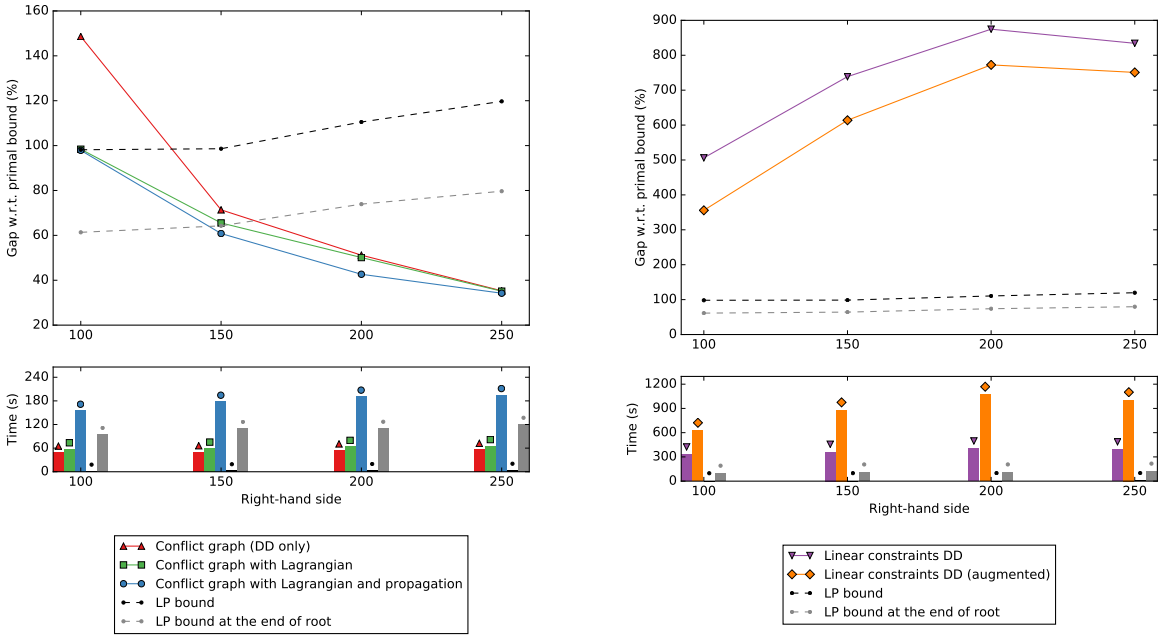


Figure 3.8: Dual bounds from relaxed decision diagrams for independent set + knapsack constraints with 1000 variables, varying the right-hand side of the knapsack constraints.

3.6 Conclusion

The main question of this chapter is how to effectively construct a relaxed decision diagram from an integer programming model. Based on our computational experience with decision diagrams, we believe that it is important for decision diagrams to leverage some structure in order to be effective, rather than approach a problem in a completely generic fashion.

We propose a framework to take advantage of a substructure of the problem, which consists of constructing a relaxed decision diagram for the substructure and applying Lagrangian relaxation and constraint propagation to take into account the other constraints. As one of the substructures, we propose the use of the conflict graph, which supports an efficient construction of a reduced decision diagram. The computational results support the notion that taking advantage of structure is important.

One observation that stands out is how the bounds scale, at least for the particular set of instances tested. It suggests that decision diagrams may work better if we are able to subdivide a problem into smaller subproblems and use them for the subproblems. In fact, a subdivision is natural in MIP solvers, which is the branch-and-bound tree. In Chapter 5, we explore this direction further by applying bounds at small nodes in a branch-and-bound tree.

Chapter 4

Cutting Planes from Relaxed Decision Diagrams

4.1 Introduction

Most general-purpose cutting planes in integer programming are based on a linear programming (LP) relaxation. However, there may be useful cuts that are difficult to reach from this relaxation. In this chapter, we explore an alternative framework to generate cuts. We generate cutting planes that define facets of the convex hull of a discrete relaxation. More precisely, we use relaxed decision diagrams to compactly represent a set of integer points that contains all feasible solutions and potentially some infeasible solutions. Then, we find cutting planes that are facet-defining for the convex hull of these points.

Relaxed decision diagrams are appealing for this purpose as they may contain information about the discrete structure of the problem that is not captured by an LP relaxation. For example, relaxed decision diagrams can produce stronger bounds than LP relaxations for problems such as maximum independent set, maximum cut, and maximum 2-satisfiability [22, 20]. They also have the advantage that the strength of the relaxation is adjustable, which enables us to control the trade-off between the strength of the cuts and the time it takes to generate a cut. Lastly, these cuts are independent from the continuous relaxation that produces the fractional point to separate, and only require a (relaxed) decision diagram for the problem at hand.

In principle, the set of feasible solutions for every bounded pure integer program can be represented by decision diagrams (see Section 2.7.2). That said, constructing small, yet strong, relaxed decision diagrams may be challenging, depending on the application. The approach of Bergman et al. [22, 20] utilizes a dynamic programming formulation of the problem for this purpose. Therefore, if strong problem-specific construction methods exist, we can take advantage of them in our generic method. For example, in our computational experiments we will use a construction method for the independent set problem that is known to produce relatively strong relaxations.

Decision diagrams have been previously applied to cut generation for integer programs by Becker et al. [11]; see also [12]. In these works, the authors represent a subset of active constraints exactly with a decision diagram. Valid cuts are then generated via a Lagrangian relaxation, using the decision diagram as an optimization oracle. Our method has some similarities to these previous works, but it differs from them in several ways. From a formal standpoint, we investigate the correspondence between decision diagrams and polar sets, which enables us to prove that the generated cuts from exact decision diagrams are facet-defining. In practice, we use relaxed decision diagrams rather than exact decision diagrams for a subset of constraints. The strength of relaxed decision diagrams can be controlled by a maximum width parameter.

These cuts, directly generated from a polar set, are called target cuts, a term coined by Buchheim et al. [27]. The main difference between their approach and ours is that they use projection to make the separation problem tractable, while we use relaxed decision diagrams.

The embedding of decision diagram-based target cuts in integer programming solvers can be realized in different ways. The most natural integration is to build a (relaxed) decision diagram for the entire problem at hand. For several combinatorial optimization problems, such decision diagrams have already been developed and can be readily used [20]. In principle, it is possible to automatically build decision diagrams for arbitrary integer programming models [12], although the associated relaxed decision diagrams may not be sufficiently strong. Alternatively, it is possible to automatically generate a decision diagram for specific substructures of problem at hand, such as clique constraints, covering constraints, or partitioning constraints. The associated target cuts would then be facet-defining for those substructures, but also valid for the overall problem.

The remainder of the chapter is structured as follows. In Section 1.2.2, we introduce the general concepts and notation regarding decision diagrams. We then review target cuts in Section 4.2. Section 4.3 provides the relationship between decision diagrams and polyhedra for binary integer programs. We introduce the target cut generation method in Section 4.4. Section 4.5 describes how to construct certificates of the dimension of faces defined by cuts. In Section 4.6, we generalize the results from binary integer programs to any bounded integer program. Section 4.7 contains extensive computational results on the independent set problem to assess the potential strength of the target cuts. Section 4.8 concludes this chapter.

In this chapter, for simplicity we will focus first on binary decision diagrams (BDDs) before discussing multivalued decision diagrams (MDDs).

4.2 Target Cuts

Before defining target cuts, we first recall a concept from convex analysis useful in cutting plane theory. The *polar set* of an n-dimensional set S is defined as

$$S^\circ = \{u \in \mathbb{R}^n : u^\top x \leq 1 \ \forall x \in S\}.$$

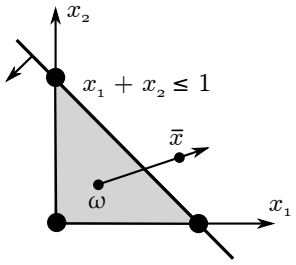


Figure 4.1: Geometrical interpretation of a target cut. A target cut defines a facet that intersects the segment between the point $\bar{x}$ to separate and an interior point ω of the polyhedron.

That is, the polar set can be viewed as the set of (coefficients of) all inequalities of the form $u^\top x \leq 1$ that are valid for S. If the origin happens to be in the interior of the convex hull of S, denoted by $\text{conv}(S)$, then the polar set corresponds to all inequalities valid for S. This is because any inequality strictly satisfied by the origin has a positive right-hand side and vice versa, and such inequality may be normalized to a right-hand side of 1.

An important property of polar sets is that each vertex of the polar set of S corresponds one-to-one to a facet of the convex hull of S, under the assumptions that $\text{conv}(S)$ is full-dimensional and the origin is in the interior of $\text{conv}(S)$. In other words, it allows us to relate the vertices of a polyhedron to its facets, and thus it is a natural channel through which we can generate valid inequalities from a discrete relaxation. For further details on polar sets, see e.g. [52].

Target cuts can be defined geometrically or algebraically [27]. Their name comes from the geometric view illustrated in Figure 4.1 and they are defined as follows. The problem we consider is to separate some set S from a point $\bar{x} \notin \text{conv}(S)$. Let ω be an interior point of $\text{conv}(S)$ and assume $\text{conv}(S)$ is a polyhedron. If we shoot a ray from the interior point ω to the point $\bar{x}$ – that is, we view $\bar{x}$ as a "target to shoot at" – then a *target cut* is an inequality defined by a facet of $\text{conv}(S)$ intersected by the ray. Observe that the ray may intersect more than one facet by hitting a lower dimensional face, and thus more than one target cut may exist for given $\bar{x}$ and ω.

From an algebraic perspective, a target cut is a valid and facet-defining cut for $\text{conv}(S)$ and has maximum violation (for a normalized right-hand side of 1) with respect to $\bar{x}$ in the translated space where ω is the origin. Such an inequality is tightly connected to polar sets: its coefficients are given by an extreme point of $(S - \omega)^\circ$ with maximum violation. Therefore, we may formally define a target cut as an inequality $u^{*\top}(x - \omega) \leq 1$ where u^* is an optimal extreme point solution for the problem $\max\{u^\top(\bar{x} - \omega) : u \in (S - \omega)^\circ\}$. This inequality is valid and facet-defining for S since it is given by an extreme point of the polar set and it cuts off $\bar{x}$ since we maximize violation. The equivalence between the geometrical and algebraic definitions is given by Buchheim et al. [27].

The choice of S is crucial: ideally we would like S to be the (integer) feasible set itself, but

optimizing over its polar set is generally impractical as it has as many constraints as (integer) feasible solutions of the problem. Therefore, we let S be some tractable projection or relaxation of the feasible set of the problem, which still yields a cut valid for the feasible set. Note that since we want to cut off $\bar{x}$, this relaxation cannot be the one that yielded $\bar{x}$ or any strictly weaker relaxation.

Buchheim et al. [27] make the separation problem for target cuts tractable by considering a sufficiently small projection of the polytope. Target cuts in this form have been computationally investigated for a number of applications: certain constrained quadratic 0-1 problems [28], the maximum cut problem [24], the equicut problem [8], and robust network design [29]. Our method differs from their approach in that we turn the cut generation tractable by using relaxed decision diagrams instead of projection.

In other previous work, Boyd [25] relates these cuts, under the name 1-polar cuts, to Fenchel cuts from a theoretical perspective. This method among others was also discussed by Cadoux et al. [30], under the assumption that an oracle for optimizing over the polar set is available. In addition, target cuts are closely related to local cuts [31], which support face certificates such as those described in Section 4.5.

4.3 Decision Diagrams and Polyhedra

The correspondence between paths and points in decision diagrams allows us to relate them to certain useful polyhedra. In this section, we first review a result from Behle [12] connecting decision diagrams to the convex hull of the set of points they represent. Then, we show that polar sets can be obtained from decision diagrams in a similar fashion.

4.3.1 Convex hull

Let S be a set of binary points of dimension n, for example the feasible set of a binary IP. Denote by $\text{proj}_x(P)$ the projection of P onto x. That is, $\text{proj}_x(P) := \{x : \exists y \text{ s.t. } (x, y) \in P\}$. Let us first write down the definition of convex hull:

$$\text{conv}(S) = \left\{ x \in \mathbb{R}^n : \exists \alpha \text{ s.t. } \sum_{\hat{x} \in S} \alpha_{\hat{x}} \hat{x} = x,\ \sum_{\hat{x} \in S} \alpha_{\hat{x}} = 1,\ \alpha \geq 0 \right\}.$$

Here, α takes the role of the coefficients through which x can be expressed as a convex combination of the points of S. In the context of BDDs, the role of α can be taken by flow variables as in the theorem below. Denote by $\delta^+(i)$ and $\delta^-(i)$ respectively the set of outgoing arcs from i and incoming arcs to i.

Theorem 4.1 (Behle [12])**.** *Consider a BDD B with vertices V and arcs A representing the set of points S, with root s and terminal t. Assume B has no long arcs. Let S_k be the set of 1-arcs at*

layer k. Define

$$\begin{aligned} P_{\textit{flow}}(B) = \{(f,x) \in \mathbb{R}^{|A|} \times \mathbb{R}^n \ : \ & \sum_{j \in \delta^-(i)} f_{ji} - \sum_{j \in \delta^+(i)} f_{ij} = 0 && \forall i \in V \setminus \{s,t\}, \\ & \sum_{j \in \delta^+(s)} f_{sj} = 1, \\ & \sum_{(i,j) \in S_k} f_{ij} = x_k && \forall \textit{ layer } k, \\ & f_{ij} \geq 0 && \forall (i,j) \in A\}. \end{aligned}$$

Then

$$\textit{proj}_x(P_{\textit{flow}}(B)) = \textit{conv}(S).$$

Compare the above formulation with the original convex hull formulation. The first two classes of constraints in $P_{\text{flow}}(B)$ indicate that f is a feasible flow of value one on the BDD B, which take the role of ensuring that the coefficients α of the convex combination sum up to one. The third constraint ensures that the total flow through the 1-arcs of each layer k is equal to x_k, which mirrors expressing x in terms of the coefficients α.

More precisely, a feasible flow of value 1 can be viewed as an edge weight vector representing the weights of a convex combination of paths from s to t. Since each path from s to t in the BDD corresponds to a point of S and vice versa, the flow can be translated to the coefficients of a convex combination of the points of S that yields x.

Note that the above theorem not only gives us a way to check if a point x is in $\text{conv}(S)$ or not, but it also allows us to explicitly express x as a convex combination of the points of S. This can be done by decomposing the flow into weighted paths.

If the BDD has long arcs, the theorem can be adapted by generalizing S_k. Let S_k instead be the set of arcs that have $x_k = 1$ as part of their labels. The rest of the theorem remains the same.

More generally, a similar characterization can also be obtained for dynamic programming formulations [48].

4.3.2 Polar set

We now relate a BDD representing a binary set of points S to the polar set of S, defined by $S^\circ = \{u \in \mathbb{R}^n : u^\top x \leq 1 \ \forall x \in S\}$. As previously discussed, polar sets are essential to the generation of target cuts.

Consider again a BDD B with vertices V, arcs A, root s, and terminal t representing S. Assume B has no long arcs for now. We define a polyhedron $P^*(B)$ and show that it formulates S° in a higher dimension.

Our model is constructed with the following intuition. First, view the constraints of the polar set as $1 - \sum_{k=1}^{n} u_k x_k \geq 0$ for each $x \in S$. That is, if we start with a budget of 1 and pay a cost of u_k whenever $x_k = 1$, we must end up with a nonnegative value.

We translate this intuition into the context of decision diagrams. Define variables v_i for each node i and u_k for each layer k. Start at s with a budget of $v_s = 1$ and traverse the BDD towards t. For each layer-k 1-arc traversed, we pay a cost of at least u_k. The value v_i at each node i must be consistent with every path leading to i. Finally, when we reach t, we must have at least $v_t = 0$ remaining for every path up to t. Note that u and v may be negative.

This intuition leads to the following formulation.

$$\begin{aligned} P^*(B) = \{(u,v) \in \mathbb{R}^n \times \mathbb{R}^{|V|} : v_j &\leq v_i - u_k && \forall \text{ 1-arc } (i,j) \text{ of layer } k, \\ v_j &\leq v_i && \forall \text{ 0-arc } (i,j), \\ v_s &= 1, \\ v_t &= 0\} \end{aligned}$$

In other words, for each path corresponding to x we pay a total of $\sum_{k=1}^{n} u_k x_k$ from a budget of 1 to traverse from s to t, so we ensure that $u^\top x \leq 1$ for all $x \in S$. We formalize this interpretation below.

Theorem 4.2. $\text{proj}_u(P^*(B)) = S^\circ$.

Proof. ($\subseteq$) Let $u \in \text{proj}_u(P^*(B))$. We want to show that $u^\top x \leq 1$ for all $x \in S$. Let $x \in S$. Then x corresponds to a path p from s to t in the BDD. For each layer-k arc (i,j) in p, we have $v_j \leq v_i$ if $x_k = 0$ or $v_j \leq v_i - u_k$ if $x_k = 1$. By summing up all these constraints, they imply $v_t \leq v_s - u^\top x$. Since $v_s = 1$ and $v_t = 0$, $u^\top x \leq 1$. Hence $u \in S^\circ$.

($\supseteq$) Let $u \in S^\circ$, that is, $u^\top x \leq 1$ for all $x \in S$. We want to show that there exists v such that the constraints of $P^*(B)$ are satisfied. Define

$$\begin{aligned} &v_s = 1,\ v_t = 0 \\ &v_j = \min \left\{ \min_{\text{1-arc } (i,j) \text{ of layer } k} \{v_i - u_k\}, \min_{\text{0-arc } (i,j)} \{v_i\} \right\} \quad \forall j \in V \setminus \{s,t\}. \end{aligned}$$

By construction of v, all constraints are satisfied except possibly the constraints for layer-n arcs (i,t). To show they are also satisfied, define $\hat{v}_t := \min\left\{\min_{\text{1-arc } (i,t)}\{v_i - u_n\}, \min_{\text{0-arc } (i,t)}\{v_i\}\right\}$, which is the definition of v_j above applied to $j = t$. It suffices to show that $v_t \leq \hat{v}_t$ since this encapsulates all arc constraints for layer n.

The definitions of v and $\hat{v}_t$ imply that there is a path p from s to t such that $\hat{v}_t = v_s - \sum_{k \in S_p} u_k$, where S_p is the set of layers in which p has a 1-arc. In other words, there exists an $x \in S$ such that $\hat{v}_t = 1 - u^\top x$. Since $u \in S^\circ$, we have $1 - u^\top x \geq 0$, and hence $\hat{v}_t \geq 0 = v_t$.

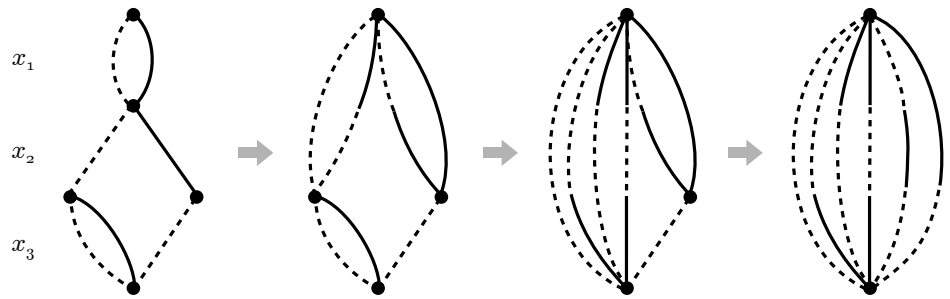


Figure 4.2: Process described in the alternative proof of Theorem 4.2: projecting out of the node variables iteratively until we reach an explicit description of the represented set. Dashed and full parts of long arcs indicate the labels of 0 and 1 respectively for the corresponding layer.

Therefore, all constraints of $P^*(B)$ are satisfied for (u, v) and thus $u \in \text{proj}_u(P^*(B))$. □

Theorem 4.2 assumes that B has no long arcs, but it can be adapted to consider them in their general form. To do so, it suffices to generalize the arc constraints in $P^*(B)$. Suppose that we have a long arc (i, j) from layer k to $k + r$ labeled with the partial solution $x_k = \ell_k, \dots, x_{k+r} = \ell_{k+r}$. Then we write its corresponding constraint in $P^*(B)$ as $v_j \leq v_i - \sum_{p=k}^{k+r} \ell_p u_p$. The first part of the above proof still holds for this variant: by summing up the arc constraints in a path representing x including the long arcs, we still obtain $u^\top x \leq 1$. The second part also holds if we generalize v_j to $\min_{\text{arc } (i,j) \text{ of label } \ell}\{v_i - \sum_{p=k}^{k+r} \ell_p u_p\}$.

In light of the idea of long arcs, we present an alternative proof for Theorem 4.2 that provides additional insight.

Alternative proof for Theorem 4.2. We consider what occurs when we apply one step of Fourier-Motzkin elimination in order to project out a variable v. Here we assume long arcs may exist.

In our context, applying Fourier-Motzkin elimination to v is reduced to summing up the constraints of each pair of incoming and outgoing arcs of v. More specifically, for each pair of incoming arc (i, v) labeled ℓ^{in} and outgoing arc (v, j) labeled ℓ^{out}, we replace their corresponding constraints by the constraint $v_j \leq v_i - \sum_{p \in R} \ell_p u_p$, where ℓ is the label ℓ^{in} concatenated with ℓ^{out} and R is the range between the layers of i and j. We may view this as replacing these arcs by a long arc with label ℓ.

Therefore, if we apply Fourier-Motzkin elimination to all nodes of the BDD except s and t, the result is a graph with one long arc for each original path with the label of the corresponding point x, as illustrated in Figure 4.2. This implies $\text{proj}_u(P^*(B)) = \{u \in \mathbb{R}^n : 0 \leq 1 - u^\top x \ \forall x \in S\} = S^\circ$. □

This proof shows us that we can replace nodes of the BDD by long arcs until we reach an explicit

description of all points through long arcs, and in practice any intermediate BDD B constructed in this process can be used to formulate $P^*(B)$. For each node v we project out, we replace $\text{in}(v) + \text{out}(v)$ arcs by $\text{in}(v) \cdot \text{out}(v)$ arcs, where $\text{in}(v)$ and $\text{out}(v)$ is the number of incoming and outgoing arcs of v respectively. In particular, we can remove any node with a single incoming arc or a single outgoing arc and reduce the number of arcs by one. We can also remove any node with two incoming arcs and two outgoing arcs without increasing the number of arcs. This process allows us to construct a more compact version of $P^*(B)$.

As previously discussed, generating target cuts requires us to work with the polar of a translated set S. We adapt Theorem 4.2 to characterize $(S-\omega)^\circ$ with BDDs for any point ω. Doing so only requires a modification of the $v_s = 1$ constraint in $P^*(B)$. Define

$$\begin{aligned} P^*_\omega(B) = \{(u,v) \in \mathbb{R}^n \times \mathbb{R}^{|V|} : \; & v_j \le v_i - u_k && \forall \text{ 1-arc } (i,j) \text{ of layer } k, \\ & v_j \le v_i && \forall \text{ 0-arc } (i,j), \\ & v_s = 1 + u^\top \omega, \\ & v_t = 0\}. \end{aligned}$$

Then, given S as the set of points represented by B,

Theorem 4.3. $\text{proj}_u(P^*_\omega(B)) = (S-\omega)^\circ$.

Proof. The proof of Theorem 4.2 can be adapted to prove this theorem. Note that $(S-\omega)^\circ = \{u : u^\top(x-\omega) \le 1 \; \forall x \in S\} = \{u : u^\top x \le 1 + u^\top \omega \; \forall x \in S\}$. In the proof of Theorem 4.2, v_s leads to the right-hand side of the constraints in the polar set. Therefore, instead of replacing v_s by 1, replace v_s by $1 + u^\top \omega$, and the same steps lead to this result. □

Theorem 4.3 is not only useful for our cut generator in this chapter, but it also provides theoretical insight on the representability of BDDs with respect to polyhedra. Define a *sub-BDD* of a BDD B as a subgraph of B structured as a BDD with the same root and terminal nodes.

Corollary 4.4. *Let B be a BDD representing S. The set of binary points of any face of conv(S) is representable by a sub-BDD of B.*

Proof. Let F be a face of $\text{conv}(S)$. Let ω be a point in the relative interior of $\text{conv}(S)$. Denote by S_ω the translation of S so that it contains the origin in its relative interior, that is, $S_\omega = S - \omega$. Additionally, consider the similarly translated $F_\omega = F - \omega$.

Consider a valid inequality $u^\top x \le v$ defining F_ω. That is, $\{x \in \text{conv}(S_\omega) : u^\top x = v\} = F_\omega$. Since the origin is satisfied by the inequality, $v \ge 0$. Without loss of generality, we consider the cases $v = 0$ and $v = 1$, since the inequality can be normalized when $v > 0$.

Suppose the inequality is of the form $u^\top x \leq 0$. Then it must be an implicit equality – in other words, $u^\top x = 0$ is valid – because otherwise the origin would not be in the relative interior of $\text{conv}(S_\omega)$. Then $F_\omega = S_\omega$, and thus $F = S$, which is representable by B.

Suppose the inequality is of the form $u^\top x \leq 1$. This implies $u \in S_\omega^\circ$. By Theorem 4.3, there exists v such that $(u, v) \in P_\omega^*(B)$. We call an arc tight if its corresponding constraint for $P_\omega^*(B)$ is tight for (u, v). That is, an ℓ-arc (i, j) at layer k is tight if $v_j = v_i - \ell u_k$ (or, if (i, j) is a long arc, $v_j = v_i - \sum_p \ell_p u_p$). We call a path tight if it is composed only of tight arcs. Let T be the set of arcs that are in at least one tight path from the root s to the terminal t. The subgraph of B induced by T forms a sub-BDD B'.

Let S' be the set of points represented by B'. Then $x \in S'$ if and only if x corresponds to a tight path from s to t, or equivalently $u^\top x = 1 + u^\top \omega$ by summing up all constraints of the path. This implies $S' = \{x \in \text{conv}(S) : u^\top (x - \omega) = 1\} = F_\omega + \omega = F$. Therefore B' represents the set of binary points of the face F. □

In other words, if we take the sub-BDD given by the paths corresponding to the points of a face, we do not end up including a path corresponding to a point not in the face.

Corollary 4.4 establishes that it is easy to represent the points of any face of a polytope corresponding to a BDD if it is easy to represent the points of the BDD itself. This can also be interpreted as a negative result: if a face of $\text{conv}(S)$ is represented by a reduced BDD with N nodes (recall that reduced BDDs are unique), then we cannot hope to construct an exact BDD representing S with less than N nodes with the same variable ordering.

As a final remark to this section, there is a good reason why Theorems 4.1 and 4.2 are similar: they are connected by duality. Given $\lambda \geq 0$, define $P_\lambda(B)$ as $P_{\text{flow}}(B)$ except that we replace the constraint $\sum_{j \in \delta^+(s)} f_{sj} = 1$ by $\sum_{j \in \delta^+(s)} f_{sj} = \lambda$. It is possible to slightly adapt Theorem 4.1 so that we have a more general result: $\text{proj}_x(P_\lambda(B)) = \lambda \, \text{conv}(S)$.

Then, given x and assuming solutions to the problems below exist, we have the following duality relationship:

$$\min\{\lambda \in \mathbb{R}_+ : \exists f \text{ s.t. } (f, x) \in P_\lambda(B)\} = \max\{u^\top x : \exists v \text{ s.t. } (u, v) \in P^*(B)\}.$$

This mirrors a duality relationship known in convex analysis: given a closed convex set C containing the origin, the gauge function of C is equal to the support function of C° – see e.g. [38]. The gauge function of a closed convex set C containing the origin is defined as $\inf\{\lambda > 0 : x \in \lambda C\}$ and the support function of a nonempty set C is defined as $\sup\{\langle s, x\rangle : s \in C\}$. In our context, this applies to $C = \text{conv}(S)$.

4.4 Target Cut Generation from Decision Diagrams

4.4.1 Cut generation algorithm

The theory developed in the previous section is useful to generate target cuts with respect to decision diagrams. Our cut generating algorithm consists of two steps:

1. construct a relaxed decision diagram B, and
2. optimize over $P^*_\omega(B)$ to generate a cut.

In this work, we assume that a relaxed decision diagram is already constructed for us and we now show that the second step yields a valid cut.

We first consider the separation problem with respect to an arbitrary polyhedron P: given a point $\bar{x}$, we would like to either find an inequality valid for P and violated by $\bar{x}$, or decide that $\bar{x} \in P$. Ideally, we would like P to be the integer hull of the problem, but that is typically intractable. We return to the choice of P after discussing how to obtain a valid cut.

The following classic result in polarity theory, discussed for instance in [52], captures an important property of polar sets.

Theorem 4.5. *Let P be a full-dimensional polyhedron containing the origin as an interior point. Then $u^\top x \leq 1$ is a facet-defining inequality of P if and only if u is an extreme point of P°.*

Moreover, given a finite set S, $\text{conv}(S)^\circ = S^\circ$ when $\text{conv}(S)$ contains the origin. This allows us to work directly with S° to generate facet-defining inequalities for $\text{conv}(S)$.

The condition on the origin can be replaced if we are given an interior point ω of P. In this case, we may translate P so that ω is in the origin and apply the above theorem. Note that we must also translate the resulting inequality back, as detailed below.

Corollary 4.6. *Let P be a full-dimensional polyhedron and let ω be an interior point of P. Then $u^\top(x - \omega) \leq 1$ is a facet-defining inequality of P if and only if u is an extreme point of $(P - \omega)^\circ$.*

Finally, we may view the above corollary from the perspective of cut generation as follows.

Proposition 4.7. *Let P be a full-dimensional polyhedron. Let ω be an interior point of P, a point $\bar{x}$ we want to separate, and*

$$u^* = \arg\max_u \{u^\top(\bar{x} - \omega) : u \text{ is an extreme point of } (P - \omega)^\circ\}.$$

Then $u^{*\top} x \leq 1 + u^{*\top}\omega$ is a facet-defining cut for $\bar{x}$ if $u^{*\top}(\bar{x} - \omega) > 1$, or $\bar{x} \in P$ otherwise.

Proof. The optimal point u^* can be attained because the polar of a set containing the origin in its interior is bounded. By Corollary 4.6, the inequality is facet-defining for P. Moreover, the inequality is a valid cut for $\bar{x}$ because we maximize the violation of the inequality with respect to $\bar{x}$, and thus a cut will be found if and only if $\bar{x} \notin P$. □

Proposition 4.7 summarizes target cut generation. We return to the choice of P. We would like P to be a tractable polyhedron that is close to the integer hull of the problem and yields valid cuts, such as a relaxation of the integer hull. Buchheim et al. [27] chooses P to be a sufficiently small projection of the integer hull.

In our algorithm, we let P be the convex hull of the set S of points represented by a relaxed decision diagram B. Any inequality valid for a relaxation of the problem is also valid for the problem itself, and therefore we may use this method to generate valid cuts if we can optimize over S°. We have shown in the previous section that $P^*(B)$ formulates S° in a higher dimensional space, and thus if the decision diagram has a small enough number of arcs, then generating a target cut should be tractable.

Therefore, the algorithm can be concisely summarized as follows. Given a relaxed decision diagram B representing S, optimize over $P^*_\omega(B)$ for some interior point ω of $\text{conv}(S)$, and the resulting point is a valid cut for $\bar{x}$ if $\bar{x} \notin \text{conv}(S)$. A more precise description is given in Algorithm 4.1, which also includes details addressed later in this section.

So far we have assumed that $\text{conv}(S)$ is full-dimensional and an interior point ω is known. We eliminate these assumptions in the following subsections. Finally, at the end of this section we discuss the degenerate case where an extreme point of $P^*(B)$ does not project down to an extreme point of $\text{conv}(S)$, required for the facet-defining property of the cut.

4.4.2 Non-full-dimensional case

We have assumed so far that $\text{conv}(S)$ is full-dimensional. Two modifications to the algorithm are required to remove this assumption: the point ω is generalized to be in the relative interior of $\text{conv}(S)$ rather than the interior, and we must handle the case when the problem is unbounded.

If the problem is bounded, the algorithm remains the same. If the problem is unbounded, then let r^* be an unbounded ray. Buchheim et al. [27] show that ${r^*}^\top(x - \omega) = 0$ is a valid equality that is violated by $\bar{x}$. Thus the algorithm may return this equality.

4.4.3 Obtaining an interior point

A relative interior point ω can be obtained from B. The geometric center of S – that is, the arithmetic mean of all points of S – can be obtained in time linear in the number of nodes of B [12], which can be used for this purpose. Note that the center of S is in the relative interior of $\text{conv}(S)$ because it can be expressed as a convex combination of all vertices of $\text{conv}(S)$ with (positive) uniform coefficients.

We present an alternative method to compute the center of S that is more suitable to our context, relative to the one proposed by Behle [12]. This may be useful to obtain an interior point for the target cut approach.

Algorithm 4.1 BDD-based target cut generation algorithm

Input: Relaxed (or exact) BDD B representing a set of points S and point $\bar{x}$ to separate.
Output: Decide whether $\bar{x} \in \text{conv}(S)$ or not. If $\bar{x} \notin \text{conv}(S)$, return a valid inequality or equality that separates $\bar{x}$ from $\text{conv}(S)$.

BDD-Target-Cut-Generator$(B, \bar{x})$
$\omega \leftarrow$ relative interior point of $\text{conv}(S)$ (e.g. the geometric center of S)
find an optimal solution or unbounded ray (u^*, v^*) to the following problem:

$$\begin{aligned} \max\ & u^\top(\bar{x} - \omega) \\ & v_j \le v_i - u_k && \forall \text{ 1-arc } (i,j) \text{ of layer } k, \\ & v_j \le v_i && \forall \text{ 0-arc } (i,j), \\ & v_s = 1 + u^\top \omega, \\ & v_t = 0. \end{aligned} \tag{P}$$

if (u^*, v^*) is an optimal solution **then**
  if ${u^*}^\top(\bar{x} - \omega) \le 1$ **then**
    return $\bar{x} \in \text{conv}(S)$ ▷ i.e., a cut cannot be derived from the relaxation B
  else
    optionally ensure u^* is an extreme point of S° (see Section 4.4.4)
    return ${u^*}^\top x \le 1 + {u^*}^\top \omega$, a cut separating $\bar{x}$ from $\text{conv}(S)$
else if (u^*, v^*) is an unbounded ray **then**
  return ${u^*}^\top x = {u^*}^\top \omega$, a valid equality separating $\bar{x}$ from $\text{conv}(S)$

The first step is to compute, for each arc (i, j), the number of paths n_{ij} from root s to terminal t that traverse (i, j). To do so, observe that $n_{ij} = n_i^- + n_j^+$, where n_i^- is the number of paths from s to i and n_j^+ is the number of paths from j to t. Calculating the former can be done in a single top-down pass: at node s we have $n_s^- = 1$, and at each node $i \neq s$, we let $n_i^- = \sum_{j \in N^-(i)} n_j^-$, where $N^-(i)$ denotes the parents of i. Likewise, the latter can be computed in a bottom-up pass: $n_t^+ = 1$ and $n_i^+ = \sum_{j \in N^+(i)} n_j^+$, where $N^+(i)$ denotes the children of i.

We then use n_{ij} to calculate the total number $N_{\ell,k}$ of paths with label ℓ at layer k. That is, $N_{\ell,k} := \sum_{(i,j) \in S_{\ell,k}} n_{ij}$, where $S_{\ell,k}$ is the set of ℓ-arcs in layer k. Since paths of B correspond to points of S, $N_{\ell,k}$ is the total number of points $x \in S$ such that $x_k = \ell$. In addition, note that $N := \sum_{(i,j) \in \delta^+(s)} n_{ij}$ is the total number of points of S. Therefore, the center of S can be expressed as $\frac{1}{N}(0N_{0,1} + 1N_{1,1}, \ldots, 0N_{0,n} + 1N_{1,n}) = \frac{1}{N}(N_{1,1}, \ldots, N_{1,n})$.

Boyd [25] argues that one must be careful when selecting the interior point because the resulting facet depends on this choice. For example, if the point is too close to a facet, then from the geometrical interpretation of target cuts we can see that the algorithm prioritizes that facet. With this intuition in mind however, the center of S appears to be a balanced choice of interior point.

This approach may not be necessary for certain problems. For example, it is easy to find an interior point for the independent set problem, as discussed in Section 4.7.1.

4.4.4 Ensuring a facet-defining cut

There is one final detail to complete the algorithm. The cut generator as described so far will produce a cut for $\bar{x}$, but it may not be facet-defining with respect to the convex hull of the points S defined by the BDD relaxation.

An extreme point of the polar set is required in order to obtain a facet-defining cut. If we use the simplex algorithm, we obtain an extreme point (u^*, v^*) of $P_\omega^*(B)$. However, u^* is not necessarily an extreme point of $\text{proj}_u(P_\omega^*(B)) = (S - \omega)^\circ$. Note that this may only happen in the degenerate case where u^* is not a unique optimal solution.

We propose two methods to obtain an extreme point of $(S - \omega)^\circ$: an exact method and a heuristic method. The former requires n invocations of an LP solver, which may make it impractical for larger instances, but it is theoretically relevant as it produces a facet-defining cut in polynomial time.

Assume that the solution is not an extreme point of $\text{proj}_u(P_\omega^*(B))$. Let V^* be the optimal value of the problem, that is, $V^* = u^{*\top}(\bar{x} - \omega)$. First note that u^* lies on the face F of $\text{proj}_u(P_\omega^*(B))$, where $F = \{u \in \text{proj}_u(P_\omega^*(B)) : u^\top(\bar{x} - \omega) = V^*\}$. We are interested in finding an extreme point of F, as it is also an extreme point of $\text{proj}_u(P_\omega^*(B))$ and it is optimal.

The following algorithm ensures that the solution is an extreme point of F. First, add the constraint $u^\top(\bar{x} - \omega) = V^*$ to the LP. For $j = 1, \ldots, n$, iteratively reoptimize the problem by maximizing u_j, and at the end of each iteration add the constraint $u_j = v^j$, where v^j is the optimal

value at iteration j. At the end of this algorithm, we obtain an extreme point of F. To see why, let F^j be F with the addition of these constraints up to iteration j. Throughout the algorithm, we maintain the property that F^j is a face of F. Since F^n must be a single point and it is a face of F, the final solution is an extreme point of F. Note that we can stop any time none of the reduced costs of the nonbasic variables are zero, since then the solution is unique and hence an extreme point of F.

The systematic method above may be too slow for efficient application in practice. A faster heuristic is as follows. As before, first add the constraint $u^\top(\bar{x} - \omega) = V^*$ to the LP. Then apply a small random perturbation to the objective with respect to the u variables and reoptimize. The resulting solution is likely to be unique since, for any polytope, optimizing over a uniform random direction yields a unique solution with high probability.

4.5 Face Certificates

Duality allows us to numerically certify that a target cut separating $\bar{x}$ from S is facet-defining for $\text{conv}(S)$ in the case S is given explicitly. If S is given implicitly in a decision diagram, we may not obtain a facet-defining cut directly from the cut generating LP as discussed in the previous section. Nevertheless, we can still find a lower bound for the dimension of the face defined by the cut and certify it.

Define a *k-face certificate* for an inequality $u^\top x \leq 1$ as a set C of k affinely independent points in S such that $u^\top x = 1$ for all $x \in C$. It certifies by definition that the face given by $\{x \in \text{conv}(S) : u^\top x = 1\}$ has dimension at least k.

Our goal is to find face certificates when S is implicitly represented as a decision diagram. Recall that the LP we solve to generate a target cut from a decision diagram B with nodes V and arcs A is $\max_{u,v}\{u^\top \bar{x} : (u, v) \in P^*(B)\}$, denoted by (P). Its dual is $\min_f\{\sum_{j\in\delta^+(s)} f_{sj} : f \in \tilde{P}_{\text{flow}}\}$, denoted by (D), where $\tilde{P}_{\text{flow}} = \{f \in \mathbb{R}^{|A|} : \sum_{j\in\delta^-(i)} f_{ji} - \sum_{j\in\delta^+(i)} f_{ij} = 0 \ \forall i \in V \setminus \{s, t\},\ \sum_{(i,j)\in S_k} f_{ij} = \bar{x}_k \ \forall \text{ layer } k,\ f \geq 0\}$.

In order to obtain face certificates, we apply flow decomposition to an optimal value of (D). Consider the following class of flow decomposition algorithms, which turns a flow f into weights α for all paths (points in S). The algorithm below, which we call *standard flow decomposition*, iteratively selects paths and absorbs all possible flow in each step.

1. Choose a path p with positive flow according to any criteria. Let e be the bottleneck arc of p: $e = \arg\min_{e'\in p} f_{e'}$. Set α_x to f_e, where x is the point corresponding to p.
2. Reduce the flow of the path p by f_e. That is, set $f_{e'}$ to $f_{e'} - f_e$ for all $e' \in p$.
3. Repeat steps 1 and 2 until the value of the flow becomes zero.
4. Set all remaining α_x to zero.

The following theorem assures that, after solving (P) to optimality, if we apply a standard flow decomposition to its dual (D) and obtain k points with positive flow, then the dimension of the face of conv(S) defined by the cut is at least k. Denote by S^+_α the set $\{x \in S : \alpha_x > 0\}$ for a given α.

Theorem 4.8. *Let (u^*, v^*) and f^* be an optimal primal-dual pair for (P) and (D). If f^* is an extreme point of $\tilde{P}_{flow}$ and α^* is the result of a standard flow decomposition applied to f^*, then $S^+_{\alpha^*}$ is a $|S^+_{\alpha^*}|$-face certificate for $u^{*\top} x \leq 1$ with respect to conv(S).*

Proof. See Appendix 4.A. □

In the remainder of this section, we consider the case where conv(S) is used as a relaxation of the problem and we would like to find face certificates with respect to the integer hull P_I of the problem rather than conv(S). In other words, we seek a flow decomposition algorithm that finds a set S^+_α with many points in P_I.

First note that finding S^+_α that maximizes $|P_I \cap S^+_\alpha|$ is NP-hard. If we applied such a method to any flow that is positive in all arcs, it would allow us to identify whether a decision diagram contains a point in P_I or not. This is NP-hard because it generalizes the 0-1 IP feasibility problem, if we consider the width-1 decision diagram representing $\{0,1\}^n$.

Instead, we use a simple heuristic criterion: at each step of the decomposition, we choose a path that minimizes the sum of violations across all constraints of the corresponding IP. Finding this path is a matter of optimizing over the BDD, which can be done efficiently.

These certificates have further practical uses other than being a measure of strength. For instance, since they can be computed relatively quickly, they may be used in a rule to determine whether to continue generating more cuts or not. Moreover, any point from a certificate for P_I is a feasible point for the problem and may be used as a primal bound.

4.6 Multivalued Decision Diagrams

Up to the previous section, we have only considered binary decision diagrams. In this section, we extend the theoretical framework and cut generator to multivalued decision diagrams (MDDs). A multivalued decision diagram is a generalization of a binary decision diagram where we allow arcs to represent any value from a finite set, not only 0 or 1. This allows us to represent the feasible set of any bounded pure integer program.

Theorems 4.1, 4.2, and 4.3 may be adapted to the case the decision diagram is multivalued. Let M be an MDD with vertex set V, arc set A, root node s, and terminal t. Let S be the set of points M represents.

We start with Theorem 4.1. Denote by D_k the (finite) domain of variable k; that is, x_k may take any value in D_k. For every layer k and $\ell \in D_k$, let $S_{k,\ell}$ be the set of arcs that have $x_k = \ell$ as

part of their labels. We generalize P_{flow} by adapting only the third class of constraints, as shown in the following formulation:

$$\begin{aligned} P_{\text{flow,MDD}}(M) = \{(f,x) \in \mathbb{R}^{|A|} \times \mathbb{R}^n \ : \ & \sum_{j \in \delta^-(i)} f_{ji} - \sum_{j \in \delta^+(i)} f_{ij} = 0 && \forall i \in V \setminus \{s,t\}, \\ & \sum_{j \in \delta^+(s)} f_{sj} = 1, \\ & \sum_{\ell \in D_k} \sum_{(i,j) \in S_{k,\ell}} \ell f_{ij} = x_k && \forall \text{ layer } k, \\ & f_{ij} \geq 0 && \forall (i,j) \in A\}. \end{aligned}$$

Then

Theorem 4.9. $\text{proj}_x(P_{flow,MDD}(M)) = conv(S)$.

Theorem 4.9 holds for the same reason as Theorem 4.1: a flow f in $P_{\text{flow,MDD}}(M)$ may be viewed as a convex combination of paths, and thus x corresponds to a convex combination of points in S. This modification only generalizes the correspondence between f and x to accept values other than 0 and 1.

We now adapt Theorem 4.2. Assume M has no long arcs for now. We generalize the polytope P^* from Section 4.3.2 as follows:

$$\begin{aligned} P^*_{\text{MDD}}(M) = \{(u,v) \in \mathbb{R}^n \times \mathbb{R}^{|V|} : \ & v_j \leq v_i - \ell u_k && \forall \ \ell\text{-arc } (i,j) \text{ of layer } k, \\ & v_s = 1, \ v_t = 0\}. \end{aligned}$$

If M has long arcs, then the same modification from Section 4.3.2 applies here: if the label of the long arc is $(\ell_k, \ldots, \ell_{k+r})$, then we generalize the term ℓu_k above to $\sum_{p=k}^{k+r} \ell_p u_p$.

Similarly to Theorem 4.2, this polytope can be projected to the space of the u variables to obtain the polar set of S.

Theorem 4.10. $\text{proj}_u(P^*_{MDD}) = S^\circ$.

The alternative proof of Theorem 4.2 from Section 4.3.2 already takes into account general labels, and thus it proves Theorem 4.10 without any changes.

Finally, let $P^*_{\omega,\text{MDD}}(M)$ be equal to $P^*_{\text{MDD}}(M)$ except that we replace the constraint $v_s = 1$ by $v_s = 1 + u^\top \omega$. The theorem below generalizes Theorem 4.3 for MDDs.

Theorem 4.11. $\text{proj}_u(P^*_{\omega,MDD}) = (S - \omega)^\circ$.

Theorem 4.11 can be shown through the same line of reasoning as in Theorem 4.3.

Given that the entire theoretical framework from Section 4.3 can be generalized to MDDs, the cut generation algorithm also follows. In order to generate a target cut from a relaxed MDD M, we apply Algorithm 4.1, except that we optimize over $P^*_{\omega,\text{MDD}}(M)$ rather than $P^*_\omega(B)$ for a BDD B.

The theorems above show that MDDs can be used to generate target cuts for general bounded IPs. However, for a given combinatorial structure, a binary formulation may work best for an IP solver, but at the same time an MDD may be more natural or compact than a corresponding BDD. We next show how we can use an MDD to generate cuts for the binary IP formulation. More precisely, for each variable x_k with value ℓ in the MDD, we define variables y_{kp} with value 1 when $p = \ell$ or 0 otherwise, for $j = 1, \dots, L$. These variables will be used to represent the binary IP model.

Let S_{bin} be the feasible set of this binarized IP. Then we may directly obtain the polar set of S_{bin}, allowing us to generate cuts for a binary IP from an MDD. Define

$$P^*_{\text{MDD,bin}}(M) = \{(u, v) \in \mathbb{R}^{nL} \times \mathbb{R}^{|V|} : v_j \leq v_i - u_{k\ell} \qquad \forall\ \ell\text{-arc } (i, j) \text{ of layer } k,$$
$$v_s = 1,\ v_t = 0\}.$$

Corollary 4.12. $\text{proj}_u(P^*_{MDD,bin}) = S^\circ_{bin}$.

Proof. We utilize the equivalence of the MDD with an associated BDD. Suppose that we replace a variable x_k by the corresponding binary variables y_{kp} for $p = 1, \dots, L$. This transformation can be viewed in the MDD as replacing each ℓ-arc by a long arc with label $y_{k\ell} = 1$ and $y_{kp} = 0$ for $p \neq \ell$. The resulting decision diagram is a BDD B with long arcs. Observe that $P^*(B)$ is exactly the polyhedron $P^*_{\text{MDD,bin}}(M)$ above. Note that since we only replaced the arc labels, the graph structures of the MDD and BDD are the same. □

Therefore, we may optimize over $P^*_{\text{MDD,bin}}(M)$ to generate cuts from an MDD M for the corresponding binary IP.

4.7 Computational Results

We computationally investigate target cuts from relaxed decision diagrams from two points of view: strength and practicality. To evaluate the strength of the cuts, we observe the objective gap closed by the cuts and the dimensions of the faces they define. We do so on smaller instances for which we can construct the exact BDD, as we are interested in how they vary with the strength of the relaxation. Next, we examine overall solving times and size of the branch-and-bound tree on larger instances to determine how practical these cuts are. This takes into consideration the time it takes to generate them. Finally, we compare them to the Lagrangian approach proposed by Becker et al. [11]. Implementation and instances are available in https://www.github.com/ctjandra/ddopt-cut.

Before presenting the computational results, we first describe the instances we select and further details of our experimental setup.

4.7.1 Instance selection

We test our cut generation algorithm on two problem classes: the maximum independent set problem and the minimum set covering problem, both unweighted. Our goal is not necessarily to improve upon the state of the art of solving these problems, but to evaluate the advantages and disadvantages of target cuts from relaxed decision diagrams.

In particular, we require strong relaxations to generate cuts that can improve the performance of a MIP solver. Thus, in order to extract meaningful observations about these cuts, we choose instances of these problem classes for which sufficiently strong relaxations exist. These are high density graphs for independent set and low bandwidth instances for set covering, which we detail in the following subsections.

Maximum independent set problem

The maximum independent set problem is to find the maximum number of pairwise nonadjacent vertices in a given graph. Previous work provides an efficient construction of a reduced BDD for the independent set problem with bounds comparable or better than LP root node bounds [22].

We use an IP formulation based on clique cover. Given a graph $G = (V, E)$, we generate a clique cover $\mathcal{C}$ of V. Each clique is generated by selecting a vertex with the largest degree and greedily including vertices with the largest degrees. The maximum independent set problem can be then formulated as $\max\{\sum_{v\in V} x_v : \sum_{v\in C} x_v \leq 1 \ \forall C \in \mathcal{C},\ x \in \{0,1\}^V\}$.

We test our cut generator on random graphs with varying density, using the Erdős-Rényi model. That is, we generate a graph $G(n, d)$, on n vertices, in which each pair of distinct vertices is joined by an edge with probability d. This probability d is also the average density of the graph. We examine random graphs with densities d equal to 50% and 80%. We find that both IP and BDDs tend to have more difficulty with graphs of smaller densities when the number of vertices is fixed. For this reason, the number of vertices of the graphs we test on depend on the density.

We omit experiments on graphs with lower densities since tests on 50% density instances already illustrate a case where relaxed BDDs are not tight enough for practical cuts. Lower density instances are less suitable for small tight BDD relaxations.

For experiments on gap closed and dimension, we are interested in how the strength of the cut changes as we vary the relaxation width. Thus, we use a set of smaller instances for which we can construct an exact BDD as a benchmark. We consider random graphs with 120 and 300 vertices of densities 50% and 80% respectively. For experiments on solving time and number of nodes, we consider random graphs with 250 and 400 vertices of densities 50% and 80% respectively. We examine average results for 10 random instances for each graph size and density.

We use a minimum degree variable ordering to construct the BDD: at each layer, we select the vertex with the smallest degree with respect to the vertices not yet selected.

Since an interior point is known for the maximum independent set problem, we use the interior

point $\omega = (\frac{1}{2n}, \ldots, \frac{1}{2n})$, where n is the number of vertices of the instance. This is an interior point because the zero vector is feasible and all unit vectors e^j are feasible, where $e^j_i = 1$ if $i = j$ or 0 otherwise. This appears to be a reasonable choice as this is a point that is close to the origin and should not be too close to any facet except the ones defined by the nonnegative constraints. Moreover, its polyhedron is full-dimensional, so we do not have to be concerned about that aspect of cut generation.

Minimum set covering problem

Given a set of elements U and a collection $\mathcal{S}$ of subsets of U, the minimum set covering is to find the smallest number of sets in $\mathcal{S}$ such that their union covers all elements in U. We use the following traditional IP formulation: $\min\{\sum_{S\in\mathcal{S}} x_i : Ax \geq 1,\ x \in \{0,1\}^{\mathcal{S}}\}$, where A is such that $A_{iS} = 1$ if $i \in S$ or 0 otherwise.

We test the target cuts on set covering instances where the constraint matrix A has low bandwidth, known to support strong relaxed decision diagrams. These instances are randomly generated using the same process from Bergman et al. [17], which creates a staircase-like structure in A. Given number of variables n, set size k, and bandwidth $b_w \geq s$, for each row i of A we select a random subset S_i of size k of $\{i, i+1, \ldots, i+b_w-1\}$, and we let $A_{ij} = 1$ if $j \in S_i$ or 0 otherwise. The number of constraints of an instance is $n - b_w + 1$.

The three groups of instances that we test on have $n = 250$ variables, set size $k = 30$, and bandwidth $b_w \in \{40, 50, 60\}$. Relaxed BDDs become weaker as the bandwidth increases, so this choice of parameters allow us to evaluate the performance of the cuts as we weaken the relaxation. We consider average results of 16 instances for each of these three groups. For the experiments on strength of cuts in which we examine the full range of widths, we only consider instances of the smallest bandwidth 40.

While a BDD can be constructed specifically for the set covering problem [19], in our implementation we use a generic BDD construction for linear inequalities using the top-down construction from Section 2.7.2, which includes checking if each constraint is infeasible or will always be feasible with respect to binary domains. To ensure a strong relaxation, the variable order in the BDD is the same as the column order in the matrix A. In practice, one may run a heuristic to find an ordering that minimizes bandwidth.

Moreover, the polyhedron is full-dimensional for this particular set of instances when $k \geq 2$, as the linearly independent points $(1, \ldots, 1) - e^j$ for all $j = 1, \ldots, n$ are feasible. We use the point $(1, \ldots, 1)$ as the origin of the target cut. Although it is not an interior point, it is still a valid point if we are not concerned about generating facets of the form $x_i \leq 1$.

4.7.2 Experimental setup

We use CPLEX 12.6 as a MIP solver. It is commonly known that the behavior of MIP solvers can vary greatly. In order to reduce this variability and emphasize the effect of the cuts, CPLEX heuristics are disabled. We also compare our cuts with CPLEX cuts separately as we are interested in the behavior of our cuts independently from other cuts. In other words, CPLEX cuts are disabled in all of the runs we generate target cuts. The presolve reduction parameter is set to linear in all experiments, which is required for user cut generation. The root LP is solved with the barrier method for independent set and the automatic setting is used for set covering. When solving the LP to generate cuts with CPLEX, we set the preprocessing aggregation limit to a high value (100), as we find that it can substantially help in solving the LP. The experiments were performed on a 2.33Ghz Linux machine with 32GB of RAM. No issues with memory usage were observed.

The merging rule of a relaxed BDD shapes the relaxation and thus it is natural is to search for one that works well specifically for generating cuts. However, in our experience, we find that the standard merging rule that merges nodes with weak objective bounds works well for cut generation. We observe in preliminary experiments that merging rules based on Euclidean and Manhattan distance from the point to separate do not perform as well as the objective-based merging rule. In addition, we examined a rule that merges nodes based on violation with respect to a given inequality. In fact, even if we give it the inequality one would obtain with an exact BDD, we do not find a substantial improvement. Therefore, our experiments use the objective-based merging rule for both problem classes.

In our experiments, we do not attempt to obtain facet-defining cuts with a method from Section 4.4.4. This is because we already obtain cuts that define high-dimensional faces and the time to run the perturbation heuristic can be non-negligible for some instances. Appendix 4.B contains further experiments using the perturbation heuristic, which show they indeed increase the face dimension fully or almost fully.

4.7.3 Strength of the cuts: Gap closed

Gap closed is a commonly used proxy for the effectiveness of a cut and it is defined as follows. If r is the objective value at the initial relaxation, b_k is the objective value after adding k cuts, and v^* is the optimal value of the problem, then the *gap closed* after k cuts is computed as $\frac{r-b_k}{r-v^*}$. We are also interested with these experiments in observing how strong the cuts are as we increase the maximum width of the BDD.

Figures 4.3 and 4.5 (left) display the gap closed as we iteratively generate more and more target cuts, using different widths for the relaxed BDD. The BDD width is relative to the exact width: for example, in "Width 20%" we use a relaxed BDD with 20% of the exact width, and in "Width 100%" we use an exact BDD. The graphs show that target cuts from relaxed decision diagrams indeed close a significant percentage of the gap even when the BDD is not exact. We see that they perform

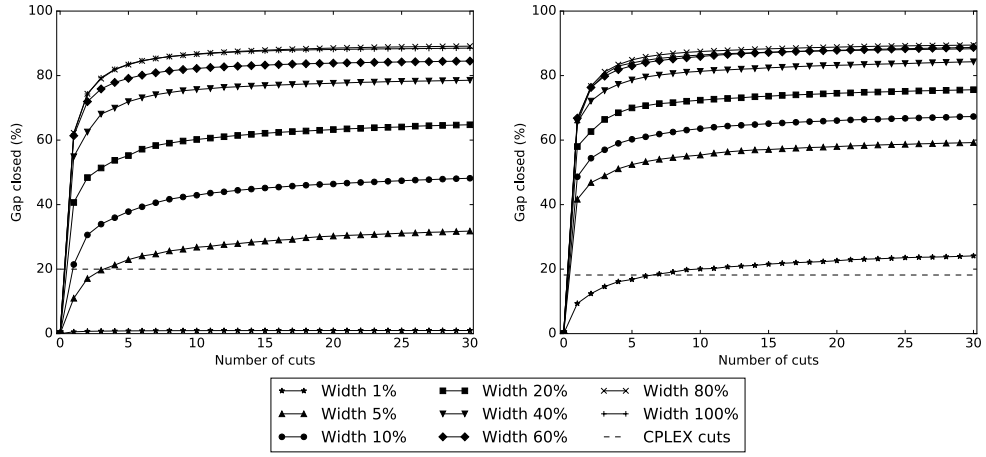


Figure 4.3: Gap closed for independent set instances of density 80% and 300 vertices (left) and density 50% and 120 vertices (right).

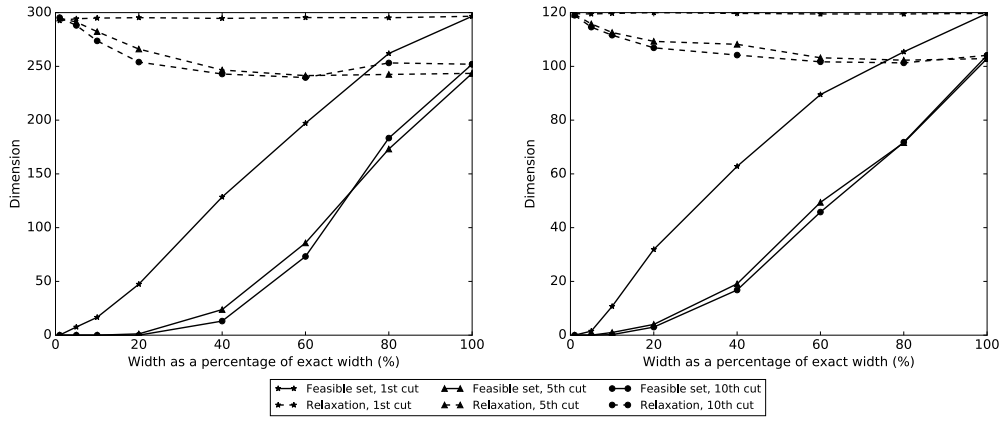


Figure 4.4: Face dimensions for independent set instances of density 80% and 300 vertices (left) and density 50% and 120 vertices (right).

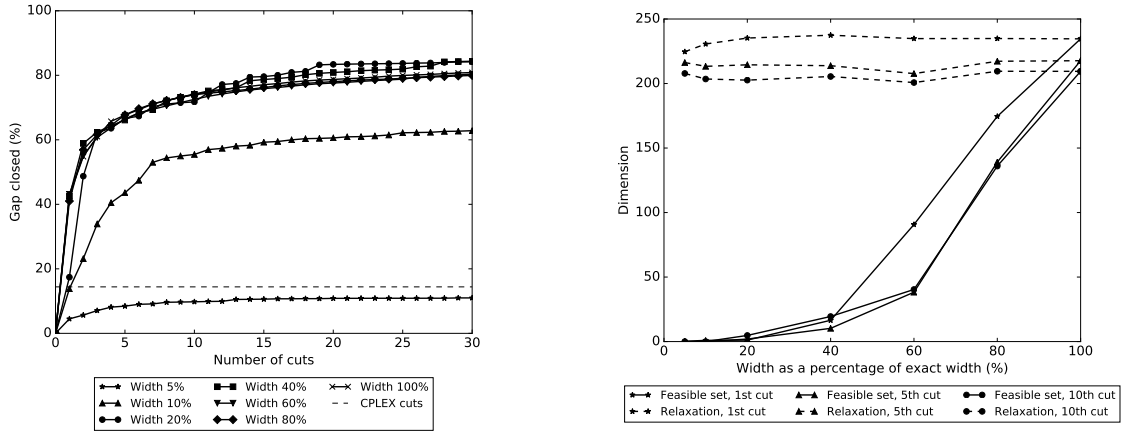


Figure 4.5: Gap closed (left) and face dimensions (right) for set covering instances of bandwidth 40.

at its best when the relaxation is tight as expected, but the graph also shows that using a high width relaxation is often as good or almost as good as using an exact BDD. In particular, for low bandwidth set covering instances (Figure 4.5, left), a cut from a BDD of width 20% covers as much gap as an exact BDD. Moreover, the gap closed for the first cut is often higher than the gap closed by CPLEX at its default settings. This suggests that we can generate effective cuts with relaxed BDDs that have a reasonable width.

Additionally, the first cuts provide a much larger gap closed when compared to later cuts. The observation that the initial cuts are stronger emerges in all of our experiments.

4.7.4 Strength of the cuts: Face dimension

A second measure of cut strength is the dimension of the face defined by a cut, with respect to the integer hull of the problem. A lower bound for these values can be obtained via flow decomposition as described in Section 4.5, using a violation minimizing heuristic in order to find a good set of feasible points. We numerically check if the points we obtain from the flow decomposition are affinely independent, in accordance to Theorem 4.8.

Figures 4.4 and 4.5 (right) exhibit these dimension bounds as the BDD width varies. We also include bounds for the BDD relaxation for context. The graphs show that increasing the width improves the dimension of the faces (of the integer hull) on these instances, which reflects the experiments on gap closed. For set covering, we do not need faces of very high dimension to close a substantial amount of gap. Additionally, we see that the initial cuts tend to have higher face dimension than subsequent cuts.

Among the instances we examined, widths of 10% for independent set and 40% for set covering were sufficient to yield nonempty feasible certificates for the first cut, meaning the cut is supporting with respect to the integer hull.

4.7.5 Overall performance of the cuts

In practice, we are interested in using these cuts to speed up MIP solvers. Total solving time, including BDD construction and cut generation, and number of nodes are reported in Figure 4.6 for the independent set problem and Figures 4.7 and 4.8 for the set covering problem. We highlight in this section independent set instances of density 80% and set covering instances of bandwidth 50 and 60, and leave graphs of the remaining instances (density 50% and bandwidth 40) to Appendix 4.B. Appendix 4.B also includes a breakdown on time spent in BDD construction, cut generation, and MIP solving.

For independent set, we observe that target cuts from relaxed decision diagrams are able to improve upon CPLEX with the first cut for density 80%. In particular, we obtain a significant decrease in the number of nodes and solving time as we increase the width of the relaxation for the first cut. On the other hand, for density 50% we cannot make a substantial conclusion on whether

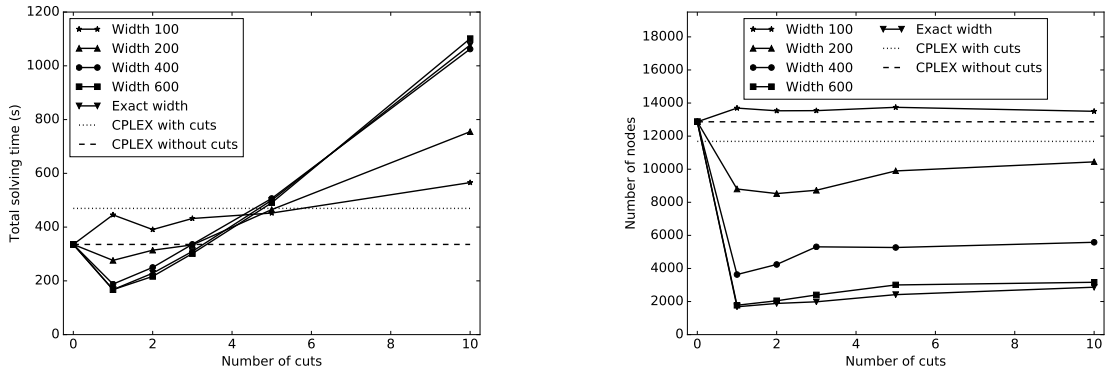


Figure 4.6: Solving time (left) and number of nodes of branch-and-bound tree (right) for cuts on independent set instances of density 80% with 400 vertices.

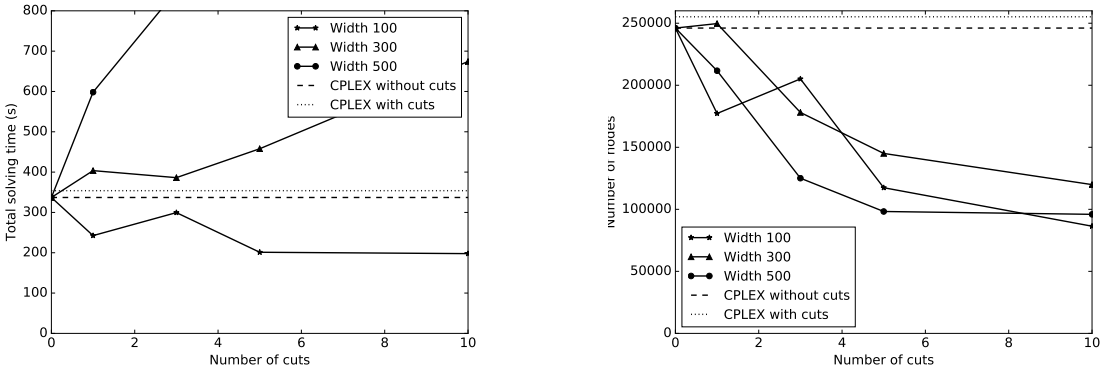


Figure 4.7: Solving time (left) and number of nodes of branch-and-bound tree (right) for cuts on set covering instances of bandwidth 50.

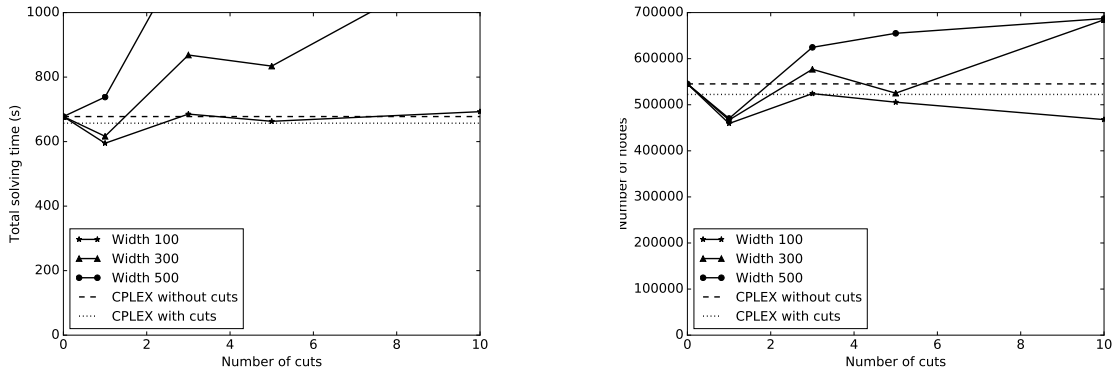


Figure 4.8: Solving time (left) and number of nodes of branch-and-bound tree (right) for cuts on set covering instances of bandwidth 60.

the cuts decrease the size of the tree, and the time it takes to generate the cut is not worth the realized speed up in the branch-and-bound tree.

For set covering, there is a substantial improvement in number of nodes for instances of bandwidth 40 and 50. In particular, target cuts for low-width relaxed BDDs improve solving time compared to CPLEX in these instances. For instances of bandwidth 60 however, the cuts are not effective since the relaxation is not strong enough.

4.7.6 Comparison with Lagrangian cuts

In this final set of experiments, we compare our cuts with a version of the Lagrangian cuts proposed by Becker et al. [11]. They are relatively fast to generate as they rely on iteratively optimizing over a BDD rather than solving a cut generating LP. The main difference is that we extract Lagrangian cuts from the same relaxed BDDs that we use to generate target cuts, while Becker et al. Becker et al. generate cuts from a BDD representing the feasible set of a fraction of active constraints. We do not apply their proposed strengthening on Lagrangian cuts since we do not do so either on target cuts. As with their version, we start with the objective as an initial solution and stop as soon as we find a valid cut.

Since the initial guess of the Lagrangian cut is the objective c, the first cut generated by the Lagrangian method is $c^\top x \leq B$ (for a maximization problem), where B is the bound given by the BDD, if B is a better bound than the root relaxation bound. This cut may be particularly helpful because it provides an upper bound for CPLEX, which can be used for pruning the tree. For a fair comparison, we add this inequality as a constraint for both cases before we start generating cuts, since it can be trivially obtained from the BDD.

For simplicity, we examine one instance group from each problem: density 80% graphs for independent set and bandwidth 50 for set covering. Figures 4.9 and 4.10 show the solving time and number of nodes in the branch-and-bound tree for target cuts and Lagrangian cuts, both with the initial bound constraint.

In the independent set case, we observe that the bound constraint from the BDD is indeed very helpful to reduce solving time. Since it is fast to generate Lagrangian cuts, it turns out that generating them is more practical in terms of total solving time than generating target cuts. We find that both target cuts and Lagrangian cuts behave similarly in terms of number of nodes, though neither provides a significant improvement with respect to the problem in which only the bound on the objective is added.

The set covering instances with low bandwidth have the unusual behavior of becoming slower if we add the bound constraint, which is why all lines start above the CPLEX lines in Figure 4.10. The Lagrangian cuts are not useful for these instances, likely in part because the cuts tend to be similar to the bound constraint, given that the method stops and returns a cut as soon as it is found. For both number of nodes and solving time, target cuts outperform Lagrangian cuts in these

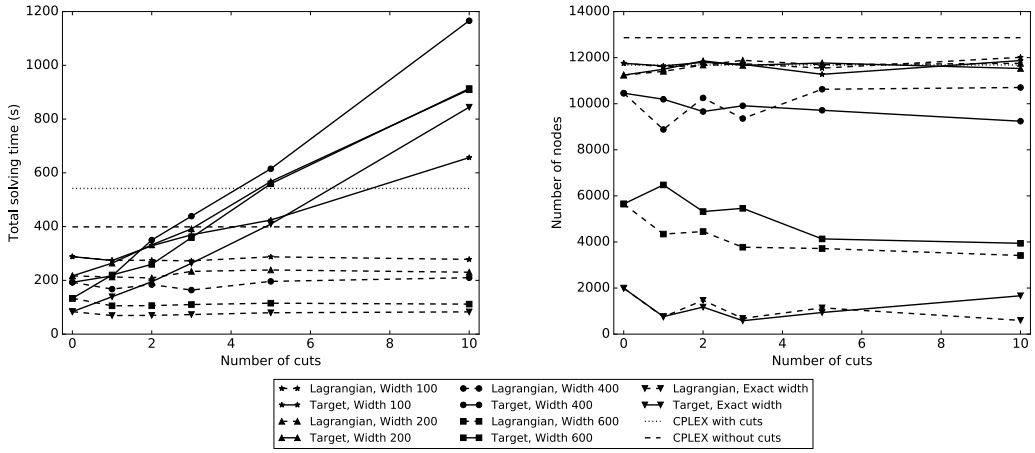


Figure 4.9: Comparison between target cuts and Lagrangian cuts from relaxed decision diagrams on independent set instances of density 80% and 400 vertices.

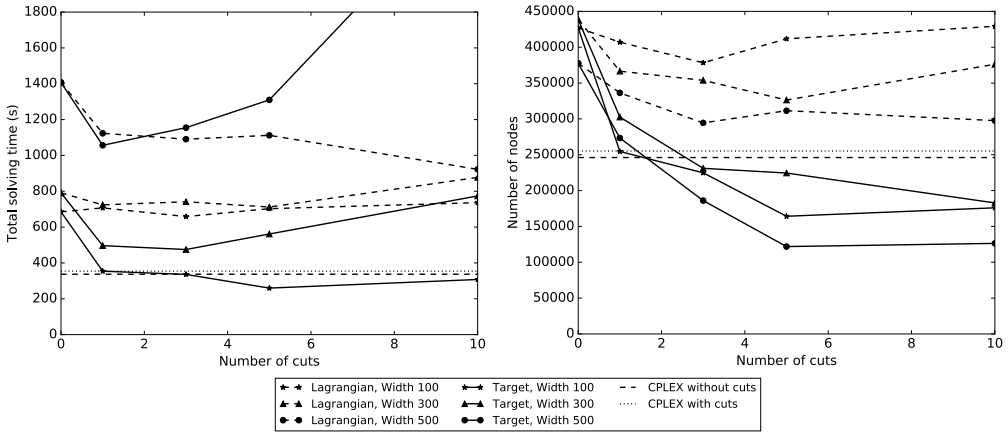


Figure 4.10: Comparison between target cuts and Lagrangian cuts from relaxed decision diagrams on set covering instances of bandwidth 50.

instances in most cases, despite being significantly more expensive to generate.

4.8 Conclusion

Previous works have shown that relaxed decision diagrams can provide good bounds, and in this work we explored an alternative viewpoint on whether they may provide good cutting planes as well. As our main contribution, we showed that generating target cuts from relaxed decision diagrams is possible due to a connection between polarity and decision diagrams.

Our experiments on the independent set problem and the set covering problem demonstrate that these cuts can be strong especially if the relaxation is tight enough. In practice, we are able to improve upon a commercial solver for certain instances, even though there is a relatively large

overhead to generate the cuts.

Nevertheless, the computational results suggest that it is worthwhile to further develop these cuts and investigate their effect on other problems, especially those that have good decision diagram relaxation but not tight continuous ones. In particular, Davarnia and van Hoeve [34] explore the use of these cuts in the context of nonlinear optimization.

The experiments discussed in this chapter focus on cuts at the root node. One of the approaches we investigate in the next chapter is the application of cuts throughout the branch-and-bound tree.

Appendix of Chapter 4

4.A Proof of Theorem 4.8 on Face Certificates

We first restate Theorem 4.8, which shows how to obtain a face certificate for a target cut from a decision diagram. All definitions required for the theorem, such as (P), (D), $\tilde{P}_{\text{flow}}$, and $S^+_{\alpha^*}$, can be found in Section 4.5.

Theorem 4.8. *Let (u^*, v^*) and f^* be an optimal primal-dual pair for (P) and (D). If f^* is an extreme point of $\tilde{P}_{flow}$ and α^* is the result of a standard flow decomposition applied to f^*, then $S^+_{\alpha^*}$ is a $|S^+_{\alpha^*}|$-face certificate for $u^{*\top}x \leq 1$ with respect to conv(S).*

We prove a series of intermediate results and the theorem will follow. For simplicity, throughout this section we assume the origin is in the interior of conv(S). At the end of this section, we argue that all of the following results still hold without this assumption.

We put aside decision diagrams for a moment and briefly discuss how to obtain certificates if we had the set of points S explicitly. In this scenario, target cuts may be generated by solving $\max_u\{u^\top \bar{x} : u \in S^\circ\}$, denoted by ($\tilde{\text{P}}$). Its dual is $\min_\alpha\{\sum_{x\in S} \alpha_x : \alpha \in P_{\text{cone}}\}$, denoted by ($\tilde{\text{D}}$), where

$$P_{\text{cone}} = \{\alpha : \tilde{S}\alpha = \bar{x},\ \alpha \geq 0\}$$

and $\tilde{S}$ is the matrix formed by the points of S in its columns. In other words, P_{cone} is the polyhedron of the coefficients of the conic combinations that express $\bar{x}$.

In the following proposition, we view a basis J of $\tilde{S}$ as a set of points of S, since $\tilde{S}$ represents a set of points of S in its columns. Moreover, we call a $\dim(S)$-face certificate a *facet certificate*.

Proposition 4.13. *Let u^* and α^* be an optimal primal-dual pair for ($\tilde{P}$) and ($\tilde{D}$).*

(i) If J is an optimal basis for α^*, then J is a facet certificate for $u^{*\top}x \leq 1$ with respect to conv(S).

(ii) If α^* is an extreme point of P_{cone}, then $S^+_{\alpha^*}$ is a $|S^+_{\alpha^*}|$-face certificate for $u^{*\top}x \leq 1$ with respect to conv(S).

Proof. Since basic variables have zero reduced costs, $x \in J$ implies $u^{*\top}x = 1$. Therefore, all points in J are tight with respect to $u^{*\top}x \leq 1$. Moreover, J is a maximal linearly independent set by definition, and thus J is a facet certificate.

If we do not have an optimal basis but α^* is an extreme point of P_{cone}, then there exists a basis associated to α^* that contains $S^+_{\alpha^*}$. Therefore, all points in $S^+_{\alpha^*}$ are tight with respect to $u^{*\top}x \leq 1$ and are linearly independent, implying $S^+_{\alpha^*}$ is a $|S^+_{\alpha^*}|$-face certificate. □

The weaker result (ii) from Proposition 4.13 is useful to prove Theorem 4.8. In the decision diagram case, we cannot obtain a facet certificate from (P) in general without modifying its optimal u^* because u^* is not guaranteed to define a facet without additional steps, as discussed in Section 4.4.4.

Our goal now is to derive a similar result for the case where S is implicitly represented as a decision diagram. That is, instead of ($\tilde{\text{P}}$), we solve (P). Recall that the dual (D) of (P) is $\min_f\{\sum_{j\in\delta^+(s)} f_{sj} : f \in \tilde{P}_{\text{flow}}\}$. In this section, it is convenient to express the constraints of $\tilde{P}_{\text{flow}}$ using matrix notation:

$$\tilde{P}_{\text{flow}} = \{f : Nf = 0,\ Vf = \bar{x},\ f \geq 0\}.$$

Here, N is the node-arc incidence matrix and V is the arc value matrix. That is, N_{ie} is 1 if arc e incides on vertex i and 0 otherwise and V_{ke} is the value that arc e assigns to variable x_k.

In order to link $\tilde{P}_{\text{flow}}$ to P_{cone}, we rely on the following polyhedron that maps a fixed $f \in \tilde{P}_{\text{flow}}$ to $\alpha \in P_{\text{cone}}$:

$$P_{\text{dec}}(f) = \{\alpha : H\alpha = f, \alpha \geq 0\},$$

where H is the arc-path incidence matrix, that is, H_{ep} is 1 if arc e is in path p or 0 otherwise. This polyhedron represents all possible ways that a fixed $f \in \tilde{P}_{\text{flow}}$ can be decomposed into $\alpha \in P_{\text{cone}}$, which is shown in the next proposition.

Proposition 4.14. *If $\alpha \in P_{cone}$, then $H\alpha \in \tilde{P}_{flow}$. Conversely, if $f \in \tilde{P}_{flow}$ and $\alpha \in P_{dec}(f)$, then $\alpha \in P_{cone}$.*

Proof. First note that all entries in the NH matrix are zero: each node has a single incoming arc and a single outgoing arc in a path, which cancel out when calculating NH. In addition, $VH = \tilde{S}$, since $(VH)_{kp}$ corresponds to the value assigned to x_k by the path p.

Let $\alpha \in P_{\text{cone}}$. Then $N(H\alpha) = 0$, $V(H\alpha) = \tilde{S}\alpha = \bar{x}$, and $H\alpha \geq 0$. Therefore, $H\alpha \in \tilde{P}_{\text{flow}}$.

Let $f \in \tilde{P}_{\text{flow}}$ and $\alpha \in P_{\text{dec}}(f)$. Then $\tilde{S}\alpha = VH\alpha = Vf = \bar{x}$ and $\alpha \geq 0$. Therefore, $\alpha \in P_{\text{cone}}$. □

Next, we prove two lemmas that together imply Theorem 4.8.

Lemma 4.15. *Let α be the result of a standard flow decomposition applied to $f \in \tilde{P}_{flow}$. Then α is an extreme point of $P_{dec}(f)$.*

Proof. It suffices to show that the columns x of $\tilde{S}$ such that $\alpha_x > 0$ are linearly independent. The reason is that, if so, there must exist a basis J of $\tilde{S}$ containing these columns. Moreover, J must be associated to α since $\tilde{S}_J\alpha_J = \tilde{S}\alpha = \bar{x}$, given that $\alpha_x = 0$ for all $x \notin J$. This implies α is an extreme point.

Let t be the number of iterations of the decomposition, or equivalently the number of points x with $\alpha_x > 0$ at the end of the decomposition. Consider a $t \times t$ matrix M where the columns (paths) are the ones with positive flow from the decomposition in the order they were encountered, and the rows (arcs) are the bottleneck arcs of each of those paths also in the same order. Then M is lower triangular since a bottleneck arc never reappears in a path after its flow is set to zero. Therefore, these columns are linearly independent. □

The following result connects extreme points of $\tilde{P}_{\text{flow}}$ and $P_{\text{dec}}(f)$ with extreme points of P_{cone}.

Lemma 4.16. *If f is an extreme point of $\tilde{P}_{flow}$ and α is an extreme point of $P_{dec}(f)$, then α is an extreme point of P_{cone}.*

Proof. Suppose for contradiction there exist $\alpha^1, \alpha^2 \in P_\alpha$ such that $\alpha = \lambda\alpha^1 + (1-\lambda)\alpha^2$ with $\lambda > 0$. We show that if α^1 and α^2 correspond to different flows, then f is not an extreme point of $\tilde{P}_{\text{flow}}$, and if they correspond to the same flow, then α is not an extreme point of $P_{\text{dec}}(f)$.

Case 1: $H\alpha^1 \neq H\alpha^2$. Define $f^t = H\alpha^t$ for $t = 1, 2$. By Proposition 4.14, $f^t \in \tilde{P}_{\text{flow}}$. Moreover, $f = H\alpha = H(\lambda\alpha^1 + (1-\lambda)\alpha^2) = \lambda H\alpha^1 + (1-\lambda)H\alpha^2 = \lambda f^1 + (1-\lambda)f^2$, with f^1 and f^2 distinct. Thus f is not an extreme point of $\tilde{P}_{\text{flow}}$, a contradiction.

Case 2: $H\alpha^1 = H\alpha^2$. It suffices to show that $\alpha^1, \alpha^2 \in P_{\text{dec}}(f)$, thus proving that α is not an extreme point of $P_{\text{dec}}(f)$. We have $f = H\alpha = H(\lambda\alpha^1 + (1-\lambda)\alpha^2) = \lambda(H\alpha^1 - H\alpha^2) + H\alpha^2 = H\alpha^2$. This also implies $f = H\alpha^1$. Therefore, $\alpha^1, \alpha^2 \in P_{\text{dec}}(f)$. □

Finally, we complete the proof of Theorem 4.8 by putting together the previous results.

Proof. Proof of Theorem 4.8. By Lemmas 4.15 and 4.16, α^* is an extreme point of P_{cone}. The result then follows from Proposition 4.13. □

As discussed in Section 4.4.4, the cut generation algorithm may involve translating points if the origin is not in the interior of $\text{conv}(S)$. While this may take away the linear independence of the points of S^+_α, they remain affinely independent, and thus all previous results still hold.

4.B Additional Computational Results

In this section, we present additional graphs on computational results. Figure 4.11 depicts solving time and number of nodes for independent set instances of density 50%, in which we see target cuts have little effect due to the relatively weak BDD relaxation. Figure 4.12 shows solving time and number of nodes for set covering instances of bandwidth 40, which are similar to the results for bandwidth 50.

Figure 4.13 gives us a breakdown of where the time is spent for the first cut for independent set instances. In both situations, the time to construct a decision diagram is very small compared to the time to solve the LP to generate a cut – less than 2%. For density 80%, we can observe that

it is not too time consuming to generate a cut and it significantly reduces the time spent in the branch-and-bound tree. This however does not happen for density 50%, where generating a cut is significantly more expensive than solving the root LP relaxation. We also see in density 80% how increasing the width of the BDD yields a stronger cut that significantly reduces the time spent in the branch-and-bound tree.

Figure 4.14 shows the same graphs from Figure 4.4, in which we examined the dimension of the faces defined by the cuts, except that in this case we apply the perturbation heuristic. If we compare them to Figure 4.4, we observe that the perturbation heuristic does as expected: it fully or almost fully increases the dimensions of the cuts with respect to the relaxation.

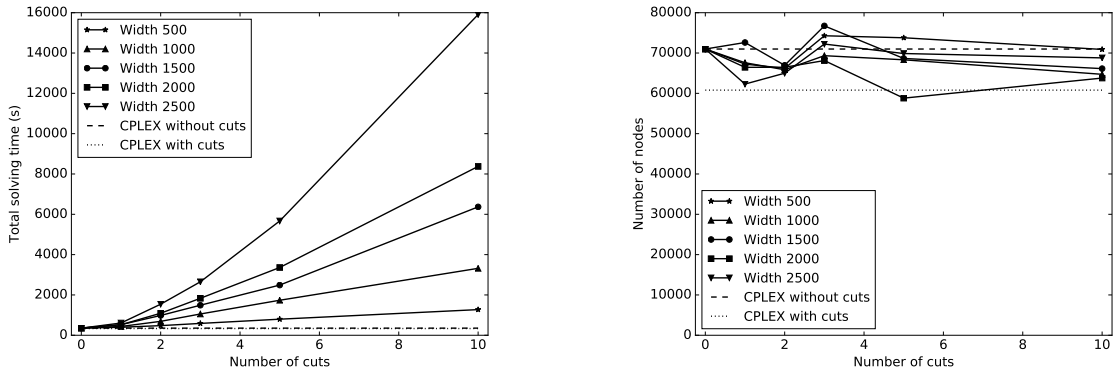


Figure 4.11: Solving time (left) and number of nodes of branch-and-bound tree (right) for cuts on independent set instances of density 50% with 250 vertices.

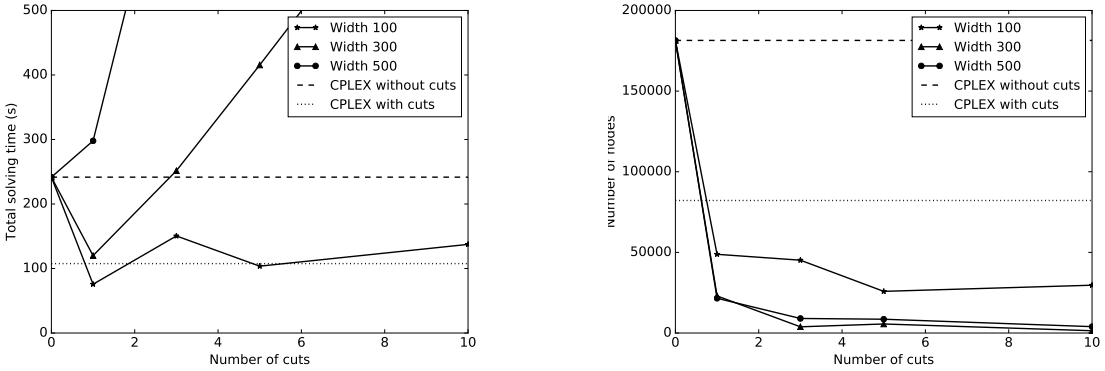


Figure 4.12: Solving time (left) and number of nodes of branch-and-bound tree (right) for cuts on set covering instances of bandwidth 40.

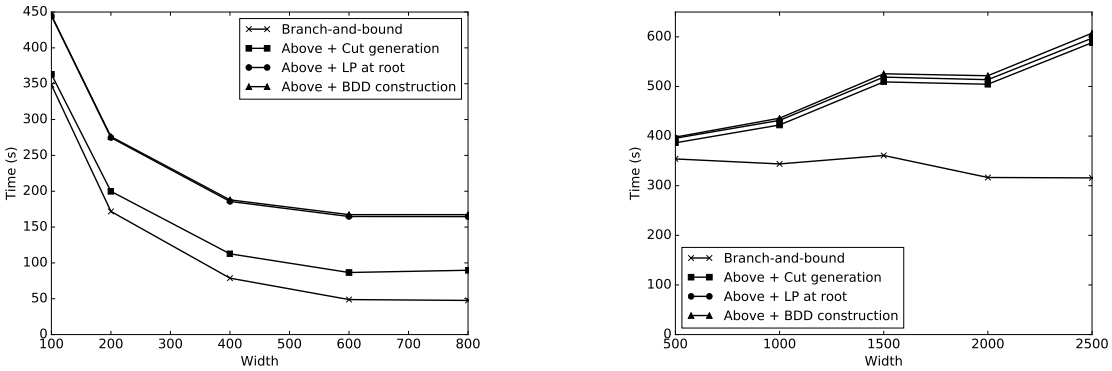


Figure 4.13: Total solving time breakdown with a single cut for independent set instances of density 80% with 400 vertices (left) and 50% with 250 vertices (right).

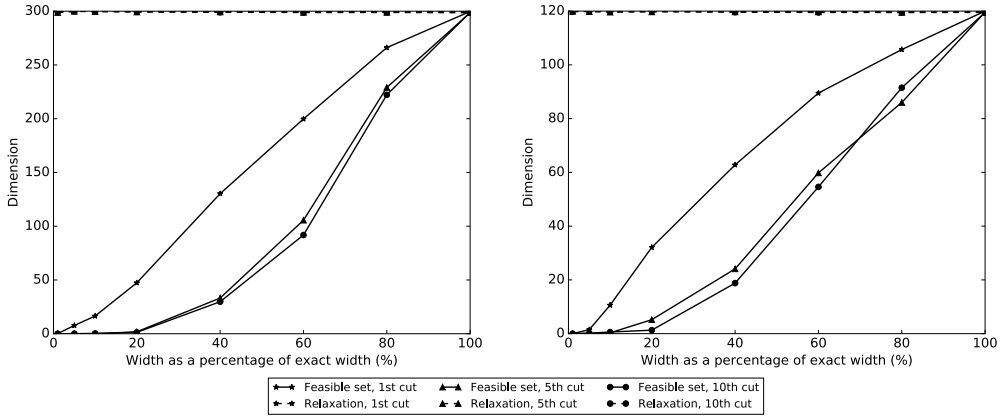


Figure 4.14: Face dimensions for independent set instances of density 80% and 300 vertices (left) and density 50% and 120 vertices (right), after applying the perturbation heuristic to increase dimension.

Chapter 5

Integrating Decision Diagrams into MIP Solving

5.1 Introduction

The previous chapters in this dissertation laid out groundwork for embedding decision diagrams into a MIP solver. We have explored how to extract useful decision diagrams from an integer programming model in Chapter 3.

Once a good relaxation is constructed, the question becomes how to use the information gathered in the decision diagram to aid the solving process. We discussed in detail one approach, cutting plane generation, in Chapter 4. In this chapter, we examine three more approaches to use decision diagrams in MIP solvers:

1. Presolve techniques with decision diagrams
2. Cut generation during MIP search
3. Primal and dual bound generation during MIP search

5.2 Presolve Techniques

An important component of MIP solvers is the presolve step, which preprocesses the model in order to
or constraints and tightening bounds or constraints. In certain problems, the effect of presolve is substantial. For a survey of presolve techniques, see [4]. We examine two presolve techniques under the perspective of decision diagrams: bound strengthening and coefficient strengthening.

In the context of decision diagrams, we will see in this section that bound strengthening can be viewed as a way to propagate information from a linear constraint to a decision diagram. Conversely,

Example: Bound strengthening for x_2 with respect to $-x_1 + x_2 \leq 0$ using the relaxed decision diagram below:

(B_{std}) $x_2 \leq -\min_{x_1 \in S_2}\{-x_1\} \implies x_2 \leq 1$

(B_{filt}) The 1-arc in the second layer is filtered.

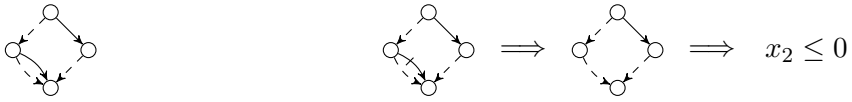


Figure 5.1: Example of the bound strengthening approaches (B_{std}) and (B_{filt}).

coefficient strengthening can be interpreted as propagating information from a decision diagram to a linear constraint.

Throughout this section, we assume that we have a relaxed decision diagram D representing the set of points S, which contains the feasible set of the problem. We also assume that we have a constraint $a^\top x \leq b$ that is valid for the problem but not valid for all points in S. For convenience, we take a variable x_j, whose bound or coefficient is to be strengthened, and we express it as $a_I^\top x_I + a_j x_j \leq b$, where I denotes the set of all variable indices except j.

5.2.1 Bound strengthening

The aim of bound strengthening is to tighten bounds of variables using a relaxation R of the problem, which could be the LP relaxation, a small subset of constraints, or simply the box induced by variable bounds. The efficiency and strength of the method respectively depend on how easy it is to optimize a linear function over the relaxation and how tight it is. If, for instance, we only use bounds as R, the procedure is very efficient but we may be able to infer little, whereas using the exact integer feasible set as R allows us to obtain the tightest bounds possible at a very expensive cost in time.

The approach proceeds as follows. Consider a constraint $a_I^\top x_I + a_j x_j \leq b$ with $a_j \neq 0$ in the model. Calculate a lower bound L for $a_I^\top x_I$ using the relaxation R projected to the space of x variables except for x_j, which we denote by R_j. That is, $L := \min_{x_I \in R_j} a_I^\top x_I$.

If $a_j > 0$, the constraint allows us to infer that $x_j \leq (b - a_I^\top x_I)/a_j$ for all feasible x. In particular, $x_j \leq (b - L)/a_j$ is valid. This results in an upper bound for x_j, which can replace the current one if it is lower. In the case $a_j < 0$, we obtain the same expression as a lower bound: $x_j \geq (b - L)/a_j$. Moreover, if x_j is integer, then the bound can be rounded up or down if it is a lower or upper bound respectively, but for simplicity we ignore rounding in this section.

Given a relaxed decision diagram corresponding to a set of points S, a natural way to use it to strengthen bounds is to treat S as the relaxation R. We denote by S_j the projection of S onto the space of x variables without x_j. In this case, L can be computed by minimizing a_I over the decision diagram, except that zero weights are assigned to the arcs of the x_j layer.

We denote this standard approach by (B_{std}). Consider now an alternative bound strengthening approach based on arc filtering, which we call (B_{filt}). Filter all arcs in the x_j layer with respect to the constraint $a_I^\top x_I + a_j x_j \leq b$. Take as lower and upper bounds the lowest and highest values of arcs corresponding to x_j. In other words, we add the bounds implied by the induced domain relaxation [39] of the decision diagram.

Arc filtering essentially performs the same type of inference that bound strengthening does, except that it is able to consider for each arc separately only the solutions using the arc. In this sense, it can be interpreted as a refined version of the standard bound strengthening. We formalize this next by showing that (B_{filt}) dominates (B_{std}).

Proposition 5.1. *The bound obtained by (B_{filt}) is at least as tight as the one obtained by (B_{std}).*

Proof. Since the lower and upper bound cases are analogous, suppose that $a_j > 0$ and we are inferring an upper bound. Assume without loss of generality that the decision diagram is ordered as $x_1, \ldots, x_n$. Let S be the set of points represented by the relaxed decision diagram given as input.

The upper bound obtained by (B_{std}) is $U_{\text{std}} := (b - \min_{x \in S_j}\{a_I^\top x_I\})/a_j$, where S_j consists of S projected to the space of all variables except x_j.

In (B_{filt}), we filter an arc (u_1, u_2) with value v_j if and only if

$$\min_{x^\uparrow \in S^\uparrow(u_1)}\{a^{\uparrow\top} x^\uparrow\} + \min_{x^\downarrow \in S^\downarrow(u_2)}\{a^{\downarrow\top} x^\downarrow\} + a_j v_j > b$$

or equivalently,

$$v_j > \frac{b - (\min_{x^\uparrow \in S^\uparrow(u_1)}\{a^{\uparrow\top} x^\uparrow\} + \min_{x^\downarrow \in S^\downarrow(u_2)}\{a^{\downarrow\top} x^\downarrow\})}{a_j}$$

where $a^\uparrow$ and $a^\downarrow$ correspond to $(a_1, \ldots, a_{j-1})$ and $(a_{j+1}, \ldots, a_n)$ respectively.

Hence, the upper bound obtained by (B_{filt}) is

$$U_{\text{filt}} := \max_{(u_1,u_2)\in E_j} \left\{ \frac{b - (\min_{x^\uparrow \in S^\uparrow(u_1)}\{a^{\uparrow\top} x^\uparrow\} + \min_{x^\downarrow \in S^\downarrow(u_2)}\{a^{\downarrow\top} x^\downarrow\})}{a_j} \right\}.$$

where E_j is the set of arcs at the layer corresponding to variable x_j.

For any $(u_1, u_2) \in E_j$, $\min_{x^\uparrow \in S^\uparrow(u_1)}\{a^{\uparrow\top} x^\uparrow\} + \min_{x^\downarrow \in S^\downarrow(u_2)}\{a^{\downarrow\top} x^\downarrow\} \geq \min_{x_I \in S_j}\{a_I^\top x_I\}$ because $S^\uparrow(u_1) \times S^\downarrow(u_2) \subseteq S_j$. Therefore, $U_{\text{filt}} \leq U_{\text{std}}$.

□

In addition, there are instances in which (B_{filt}) yields a strictly better bound than (B_{std}), as illustrated in Figure 5.1. In this example, we want to strengthen the bound of x_2 with respect to $-x_1 + x_2 \leq 0$ using a decision diagram D representing $\{(0,0),(0,1),(1,0)\}$. Applying (B_{std}) provides no improvement, since x_1 can take a value of 1 according to D, and thus the approach

implies a redundant bound $x_2 \leq 1$. On the other hand, applying (B_{filt}) removes the point $(0, 1)$ from D, leaving it with the points $\{(0, 0), (1, 0)\}$. From this, we derive $x_2 \leq 0$.

5.2.2 Coefficient strengthening

Coefficient strengthening can be viewed as a form of propagating information from (a relaxation of) the feasible set to a linear constraint. Its approach is similar to bound strengthening and relies on a relaxation R of the problem, which can take any form as discussed in the previous section. We would like to strengthen a constraint $a_I^\top x_I + a_j x_j \leq b$ and we assume x_j has lower and upper bounds l_j and u_j respectively. We again use R_j to denote the projection of R onto the space of variables without x_j.

In this section, we describe how coefficient strengthening can be used with relaxed decision diagrams. Furthermore, we provide an improvement to the coefficient strengthening technique from [51] (see also [7, 4]). This extension is particularly suitable for use with relaxed decision diagrams: for traditional relaxations it would typically require substantially more computational effort, but not for relaxed decision diagrams.

We first rederive the coefficient strengthening technique not only for purposes of completeness but also as a step towards our extension. Our description is similar to the one in [4], except for differences in notation and derivation. In particular, in the following derivation, we view the approach from the geometric perspective of tilting. This is illustrated by Figure 5.2.

In this tilting perspective, we first fix a *hinge*, an $(n-1)$-dimensional set contained in the hyperplane defining the inequality. Any tilted inequality must contain all points in the $(n-1)$-dimensional hinge, and thus a single degree of freedom is left for the inequality to tilt.

Not only we would like to avoid removing feasible solutions by tilting, but also we want the resulting inequality to dominate the original one. In other words, every solution satisfied by the new inequality but not by the original one must already be cut off by some other inequality present in the model. In order to ensure this, given any variable x_j, the coefficient strengthening technique (implicitly) tilts an inequality using as hinge the points at a bound of x_j. More precisely, if H is the hyperplane defined by the inequality, we use as hinge either $H \cap \{x \in \mathbb{R}^n : x_j = u_j\}$ or $H \cap \{x \in \mathbb{R}^n : x_j = l_j\}$. This, along with the correct tilting direction, guarantees that a successfully tilted inequality will dominate the original one within the region defined by the bounds of the variable.

For simplicity, we derive the strengthening using the upper bound hinge.[1] Tilting using the lower bound hinge can be derived analogously (see [4] for the resulting inequality).

Proposition 5.2. *Consider a valid linear constraint $a_I^\top x_I + a_j x_j \leq b$ and let $l_j \leq x_j \leq u_j$. Assume $l_j < u_j$. Let R be a relaxation of the problem and compute $U := \max_{x_I \in R_j}\{a_I^\top x_I\}$.*

[1] In the notation from [4], d is equivalent to $a_j - \hat{a}_j$ in Proposition 5.2. Its assumption $a_j \geq d$ is equivalent to the condition $U - b + a_j u_j \geq 0$ in Proposition 5.2. As shown in Proposition 5.2, coefficient strengthening is also possible when $U - b + a_j u_j < 0$ if a lower bound for x_j exists, despite not mentioned in [4].

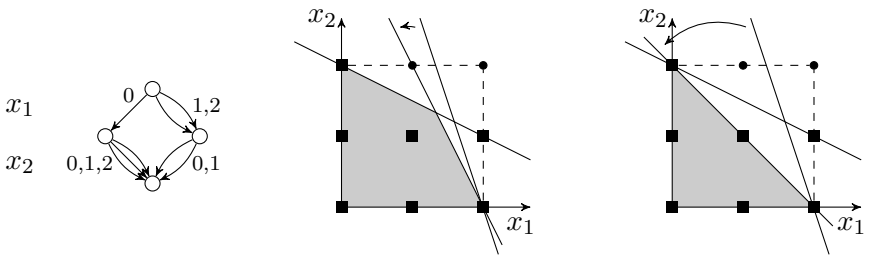


Figure 5.2: Examples for the two coefficient strengthening approaches given by Propositions 5.2 (left polyhedron) and 5.3 (right polyhedron). The square points correspond to the points in the relaxed decision diagram illustrated on the left, which are used to calculate the bounds for coefficient strengthening. The constraints are $3x_1 + x_2 \leq 6$ and $x_1 + 2x_2 \leq 4$ and the variables have lower and upper bounds 0 and 2. In this example, the constraint $3x_1 + x_2 \leq 6$ is being strengthened with respect to x_1. From a tilting perspective, the hinge is the point $(2, 0)$. On the left, the resulting constraint is $2x_1 + x_2 \leq 4$. On the right, the resulting constraint is $x_1 + x_2 \leq 2$.

Then the constraint $a_I^\top x_I + \hat{a}_j x_j \leq \hat{b}$ is valid for the problem, where

$$\hat{a}_j := \begin{cases} U - b + u_j a_j & \text{if } U - b + u_j a_j \geq 0 \\ (U - b + u_j a_j)/(u_j - l_j) & \text{otherwise} \end{cases}$$
$$\hat{b} := b - u_j(a_j - \hat{a}_j).$$

This constraint dominates the original constraint within the variable bounds if $\hat{a}_j < a_j$.

Proof. The hyperplane $a_I^\top x_I + \hat{a}_j x_j = \hat{b}$ defined by the tilted inequality must contain the hinge $\{x \in \mathbb{R}^n : a_I^\top x_I + a_j x_j = b,\ x_j = u_j\}$, which is equivalent to $\{x \in \mathbb{R}^n : a_I^\top x_I = b - a_j u_j, x_j = u_j\}$. Intersecting all equalities, we obtain the condition $\hat{b} = b - u_j(a_j - \hat{a}_j)$. In other words, the tilted inequality has the form $a_I^\top x_I + \hat{a}_j x_j \leq b - u_j(a_j - \hat{a}_j)$ and choosing $\hat{a}_j$ defines how much we tilt the inequality. In particular, if we decrease $\hat{a}_j$ relative to a_j, we are tightening the constraint within the variable bounds (since the slope with respect to x_j is increased and the hinge is at the upper bound of x_j); otherwise we are relaxing the constraint.

We want to choose an $\hat{a}_j$ that does not result in a feasible point being cut off. When $x_j = u_j$, this is guaranteed because the original linear constraint is valid and the hinge satisfies $x_j = u_j$. When $l_j \leq x_j \leq u_j - 1$, it suffices to ensure that $a_I^\top x_I \leq U$ is implied by the inequality. Using the hyperplane equality, this is equivalent to satisfying $b - a_j u_j + \hat{a}_j(u_j - x_j) \leq U$ for all $x_j = l_j, l_j + 1, \ldots, u_j - 1$. We choose the smallest $\hat{a}_j$ satisfying these conditions, that is, $\hat{a}_j = \max_{x_j \in \mathbb{Z}, l_j \leq x_j \leq u_j - 1}\{(U - b + a_j u_j)/(u_j - x_j)\}$. If $U - b + a_j u_j \geq 0$, then the maximizer is $x_j = u_j - 1$ and thus $\hat{a}_j = U - b + a_j u_j$. Otherwise, the maximizer is $x_j = l_j$ and hence $\hat{a}_j = (U - b + a_j u_j)/(u_j - l_j)$. □

We can use a relaxed decision diagram as our relaxation R, as the bound U can be computed by maximizing a_I over the decision diagram, using zero weights on the layer corresponding to x_j.

Figure 5.2 (left) shows an example of this technique applied with respect to a relaxed decision diagram. In addition, it depicts a situation where the inequality could be further tightened. We next show how this can be done at the cost of the additional computational effort of optimizing $a_I^\top x_I + a_j x_j$ over R restricted to every value v in the domain of x_j (except u_j).

Proposition 5.3. *Consider a valid linear constraint $a_I^\top x_I + a_j x_j \leq b$ and let $l_j \leq x_j \leq u_j$. Assume $l_j < u_j$. Let R be a relaxation of the problem and compute $U_v := \max_{x \in R : x_j = v}\{a_I^\top x_I + a_j x_j\}$ for all $v = l_j, l_j + 1, \ldots, u_j - 1$.*

Then the constraint $a_I^\top x_I + \hat{a}_j x_j \leq \hat{b}$ is valid for the problem, where

$$\hat{a}_j := \max_{x_j \in \mathbb{Z}, l_j \leq x_j \leq u_j - 1} \{(U_{x_j} - b + a_j u_j)/(u_j - x_j)\}$$
$$\hat{b} := b - u_j(a_j - \hat{a}_j).$$

This constraint dominates the original constraint within the variable bounds if $\hat{a}_j < a_j$.

Proof. The proof proceeds exactly the same as in Proposition 5.2, except that to ensure validity, instead of satisfying $a_I^\top x_I \leq U$, we choose the tightest $\hat{a}_j$ that satisfies the stronger conditions $a_I^\top x_I + a_j x_j \leq U_v$ for each $x_j = v$. Using the same approach as in the previous proof, this gives us $\hat{a}_j = \max_{x_j \in \mathbb{Z}, l_j \leq x_j \leq u_j - 1}\{(U_{x_j} - b + a_j u_j)/(u_j - x_j)\}$. □

Figure 5.2 illustrates the difference between the two approaches. Note that if the domains are binary, the two techniques are the same.

Optimizing over all U_v can be expensive when the domains are not small. However, using a relaxed decision diagram as R enables us to efficiently compute U_v not only for all values v, but also for all variables x_j. Computing them can be done analogously to calculating bounds in arc filtering. For each variable x_j, we can express U_v as $\max_{(u_1,u_2) \in E_{jv}}\{\max_{x^\uparrow \in R^\uparrow(u_1)}\{a^{\uparrow\top} x^\uparrow\} + v + \max_{x^\downarrow \in R^\downarrow(u_2)}\{a^{\downarrow\top} x^\downarrow\}\}$ where E_{jv} is the set of arcs that assign v to x_j. These optimal values for the partial solution set and the completion set can be computed for all nodes simultaneously in a top-down pass and a bottom-up pass respectively.

With the bounds at hand, we choose a variable to strengthen and apply the strengthening. After each strengthening step, we update the bounds according to the new coefficient, which can be done by performing a top-down and bottom-up pass starting at the layer in which the coefficient was altered, and repeat the process for a new variable. Note that the final constraint may be different depending on the order of the chosen variables.

As a final remark, an inequality can be further tilted if we refine the relaxed decision diagram. The point that is stopping the inequality from being tilted further is the optimal solution of the bound U_v such that v maximizes the expression in the definition of $\hat{a}_j$. We can check if this point is feasible with respect to the overall problem. If it is, then we know for sure that the inequality

cannot be tilted further, and furthermore we obtain a primal feasible solution. If it is not, we can remove this point from the decision diagram and repeat.

5.3 Cutting Planes in Branch-and-Bound

In Chapter 4, we have tested the use of target cuts at the root node of the branch-and-bound tree. In this section, we evaluate their effectiveness if we add them throughout the tree instead of only at the root. The cuts generated at subproblems with a small number of variables should be strong, as the relaxed decision diagrams themselves should approximate the problem well. The trade-off however is that the cuts will only have a local effect.

Given that in Chapter 4 we find that the first cut is typically the most effective, we generate a single cut at every node that has at most k variables for some threshold k that we vary. We test on the same independent set instances we use on Chapter 4: random graphs of densities 50% and 80%. In particular, they have 250 and 400 vertices respectively. We take averages of 10 instances using the shifted geometric mean[2] with factors of 10 for solving time and 100 for number of nodes.

Figures 5.3 and 5.4 show the results of this experiment. While the cuts make an impact on the number of nodes, they typically do not result in a significant improvement in solving time. Nevertheless, it does not appear to be important in these instances to generate cuts for larger subproblems, especially if the width is small. For instance, in the density 50% case with width 100, the reduction in number of nodes is roughly the same whether the threshold is 100 or 250.

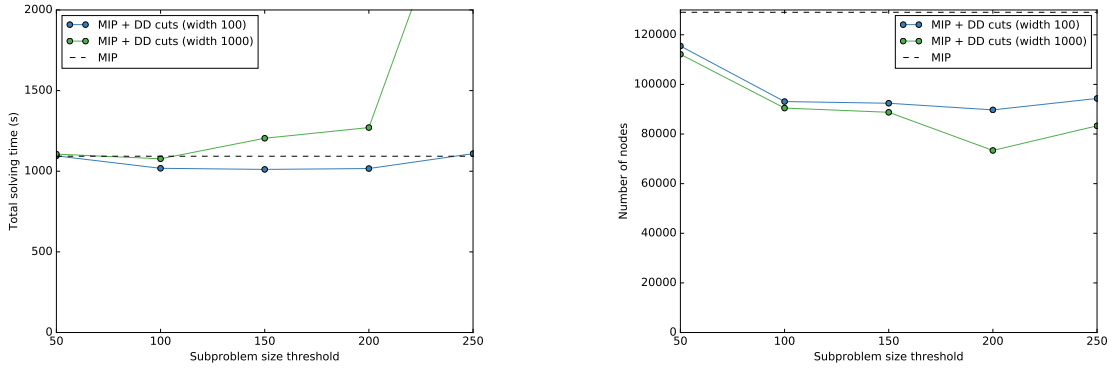


Figure 5.3: Total solving time and number of nodes when adding target cuts from relaxed decision diagrams within the branch-and-bound tree, for random graphs of density 50% and 250 vertices. Cuts are only added if the size of the subproblem is below the given threshold.

[2]The shifted geometric mean of $v_1, \ldots, v_n$ is defined as $(\prod_{i=1}^{n}(v_i + s))^{1/n} - s$ for some shift factor s.

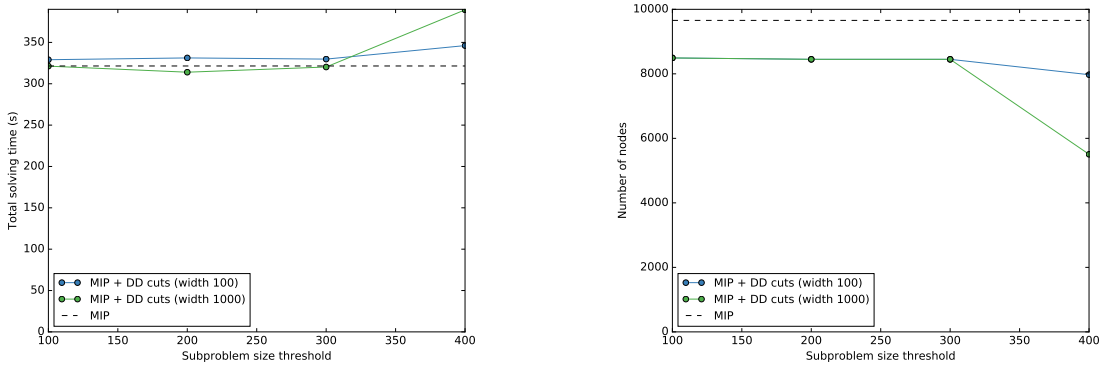


Figure 5.4: Total solving time and number of nodes when adding target cuts from relaxed decision diagrams within the branch-and-bound tree, for random graphs of density 80% and 400 vertices. Cuts are only added if the size of the subproblem is below the given threshold.

5.4 Bounds from Decision Diagrams in Branch-and-Bound

In Chapter 3, we have discussed the construction of relaxed decision diagrams aimed at generating dual bounds. It is left to address how they can aid the search process in MIP solving. In this section, we computationally investigate the natural approach of using these bounds to help prune nodes in the branch-and-bound tree.

Along with dual bounds, we also generate primal feasible solutions to improve the search process. Not only better primal feasible solutions are informative for the user if the solving process is terminated before reaching optimality, but also they can improve node pruning in the branch-and-bound tree. Conversely, primal bounds generated from the MIP solver can help speed up the construction of relaxed decision diagrams.

The computational experiments in this section are performed in two contexts. In the first, we assume that the entire problem admits effective relaxed decision diagrams. In the second, we follow the framework discussed in Chapter 3: we construct relaxed decision diagrams for a selected substructure and incorporate further constraints via Lagrangian relaxation and constraint propagation.[3]

5.4.1 Primal bounds

In this framework, primal bounds can not only be generated from decision diagrams for the MIP solver, but also primal bounds from the MIP solver can benefit decision diagram construction.

Primal feasible solutions (and thus primal bounds) can be generated from restricted decision diagrams [18], which encode a subset of feasible solutions. However, not only constructing further

[3] An early version of these computational results appeared in [56], in which an older version of SCIP (3.2.0) and a different ordering rule (minimum degree) were used.

decision diagrams for primal bounds can be inefficient, but also they require all constraints to be considered in the construction.

Instead, we present simple heuristic approaches to identify feasible solutions from the relaxed decision diagrams we use. We consider two cases: one when generic constraints are not present and one when they are. In the latter case, we assume that we use Lagrangian relaxation as described in the framework from Chapter 3.

1. **Without generic constraints.** During the construction of a relaxed decision diagram, we keep track of nodes that have been merged due to relaxation. We then find the optimal path that does not contain any of such nodes. This path corresponds to a feasible solution to the overall problem because it only contains exact nodes.

2. **With generic constraints.** The process of solving the Lagrangian relaxation problem typically involves optimizing over the decision diagram a number of times. The solutions obtained in this process are called primal iterates. For every such solution generated, we check its feasibility with respect to the overall problem. If we find that it is feasible, we store it as a primal feasible solution. This simple approach has been suggested in early works on Lagrangian relaxation [36].

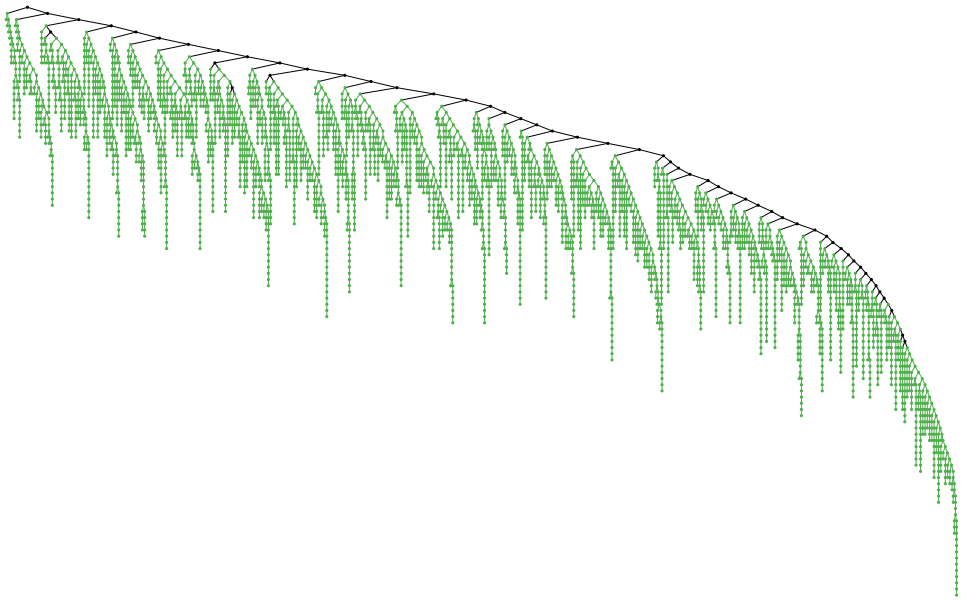
Conversely, primal bounds from the MIP solver can help eliminate solutions from the decision diagrams, potentially making them smaller. If we have a primal bound B_p and we are maximizing an objective c, then we can effectively add the constraint $c^\top x \geq B_p$ to the decision diagrams. In order to keep the size of the decision diagram in check, we do so by applying arc pruning with respect to this constraint, as described in Section 2.8.2.

5.4.2 Selecting Nodes in Branch-and-Bound

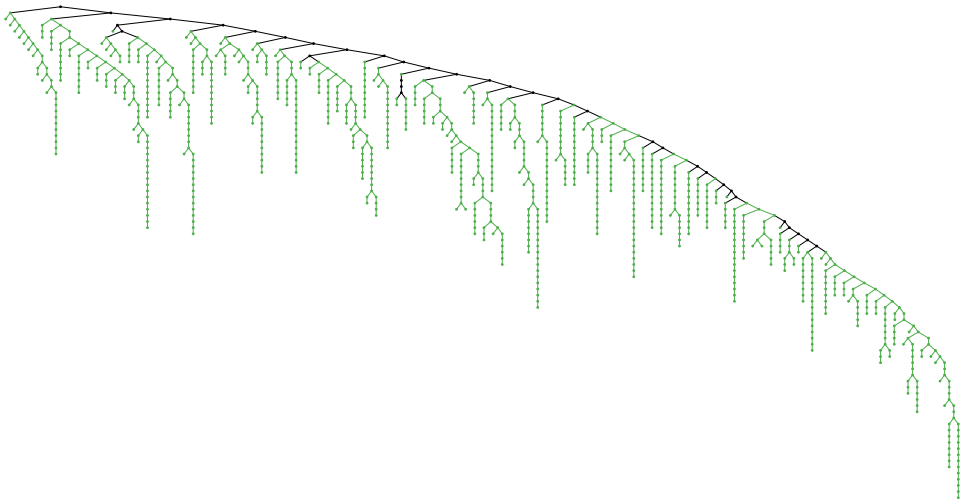
Decision diagrams only scale well when problem structure is conserved during the scaling. As an example, decision diagrams perform well for the independent set problem when the bandwidth of the graph is small. In particular, they scale well when the bandwidth is kept constant as the graph size increases, but not as well when the bandwidth increases along with the size of the graph.

In general, we cannot expect this to occur for an arbitrary problem. In particular, as evidenced by the experiments from Chapter 3, we find that decision diagrams perform better for smaller problems. As further evidence, Figure 5.5 illustrates the effectiveness of the bound over all nodes within a branch-and-bound tree for the classes of instances we discuss in the next section. We generate bounds at every node of the branch-and-bound tree and check if they would prune the node, without applying them. We observe that they perform well for small subproblems.

In our experiments, we test the simple approach of applying decision diagrams when the number of variables is below a given threshold. The threshold is a parameter to be tuned, for instance



(a)



(b)

Figure 5.5: Branch-and-bound trees for two instances: (a) independent set for a random graph of density 50% and size 150, and (b) independent set + knapsack constraints with 300 variables (see next section for definition of instances). Green nodes are those in which bounds from width-100 relaxed decision diagrams would result in pruning if applied (bounds are generated but not applied). The bounds are generated using all features: primal bounds, primal pruning, Lagrangian relaxation, and constraint propagation. For purposes of visualization, we omit nodes in which pruning or integrality is inferred by the LP.

by examining trees such as those of Figure 5.5. More sophisticated approaches are left for future research.

5.4.3 Computational experiments

We consider both the case when the entire problem is amenable to decision diagrams and when a suitable structure is only partially present. We use problem classes examined in previous chapters.

Independent set constraints

We consider random graphs parameterized by size n and density d following the Erdős–Rényi model $G(n, d)$: each edge of a graph with n vertices is included with probability d. We select two instance sizes n, 150 and 300, and vary the density d parameter from 10% to 90% in increments of 10%. For each of these parameters, we generate 16 instances. All results are shifted geometric means among these 16 instances with a shift factor of 10 for solving time and 100 for nodes. We set a time limit of one hour.

Except when otherwise stated, we generate bounds for every subproblem with at most 2/3 of the variables – that is, 100 and 200 for the instances of sizes 150 and 300 respectively. This subproblem size is manually tuned: we performed some computational experiments with different sizes and selected one that performed well for these runs. We elaborate more on subproblem size later in this section.

In terms of implementation, we construct decision diagrams based on the conflict graph of the problem, following the framework in Chapter 3, even though we know these are independent set instances. Since the DP formulation of the conflict graph generalizes the formulation for the independent set problem, the resulting decision diagrams are the same. The difference between this version and one specific to the independent set problem is overhead in time, from for instance extracting and processing conflict constraints. This is the same code we use in the experiments with generic constraints in the next subsection.

We use the primal bound-based arc pruning and the primal heuristic without generic constraints described in Section 5.4.1.

Overall performance. The first plots in Figures 5.6 and 5.7 show the overall performance of the method. For random graphs, solving maximum independent set problems tends to be easier for either dense graphs or very sparse graphs, and this is evident from the plot. The graph suggests that this approach is more beneficial around middle ranges of density, reaching about an order of magnitude in a best case scenario of density 50% and 300 vertices. The bounds do not perform well for low densities, which is consistent with observations from previous experiments such as those in Chapter 4 and [22].

Subproblem size. Figure 5.8 shows what happens if we choose a different subproblem size threshold. Selecting a larger threshold – that is, applying decision diagram bounds more often – can be helpful when we know relaxed decision diagrams are strong, such as with high-density cases. However, that can waste time when the relaxations do not scale well, as illustrated by the low-density cases, in which case focusing on smaller subproblems performs better.

We omit the plot for instances of size 300 as it depicts a similar behavior as above (although it cannot be observed for lower densities due to the time limit).

Primal bounds. Figure 5.9 illustrates the impact of primal bounds. It provides the ratio between the solving time without a given feature (primal pruning and/or primal heuristic) and the solving time with all features. The effect of the primal techniques becomes more significant with larger branch-and-bound trees. Primal pruning is particularly helpful to avoid wasting time on small nodes that are clearly infeasible.

Table 5.1 exhibits solving times for instances from the DIMACS maximum clique benchmark set [35], converted to maximum independent set. We present only the set of instances that were solved to optimality within one hour with SCIP 5.0.1. The bounds from decision diagrams are applied at every branch-and-bound node with at most 3/4 of the total number of vertices with a width of 100.

While the number of instances in Table 5.1 is small, it reflects our observation from the random graph experiments that bounds aid the solution process when the density of the graph is not small, but not otherwise. One exception is `p_hat300-2`, in which the bounds do not improve the solving time despite reducing the size of the branch-and-bound tree. It suggests that improvements in performance can be extended to more structured instances.

These computational results are generally positive; however, our main goal is to tackle more general instances. We next move on to problems with independent set constraints as a substructure.

Independent set and knapsack constraints

The instances we consider in this section are a combination of independent set (set packing) constraints with knapsack constraints. They were defined in Section 3.5, but we reiterate their description here for completeness. The integer programming model is given by:

$$
\begin{aligned}
\max\ & c^\top x \\
& \sum_{j \in C} x_j \leq 1 \qquad \text{for all } C \in \mathcal{C} && \text{(set packing)} \\
& \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for all } i = 1, \dots, m_{\text{knap}} && \text{(knapsack)} \\
& x \in \{0,1\}^n
\end{aligned}
$$

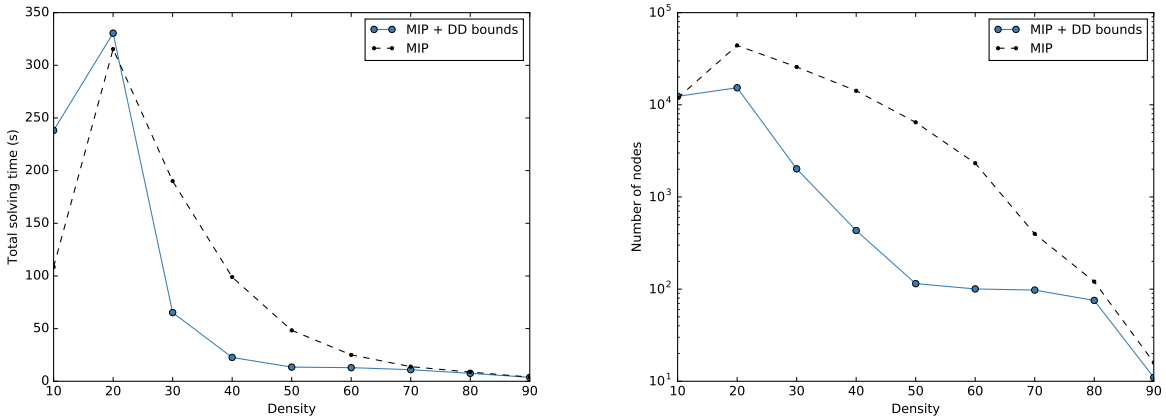


Figure 5.6: Comparison in solving time and branch-and-bound tree size between applying bounds from decision diagrams and not applying them in independent set instances with 150 vertices.

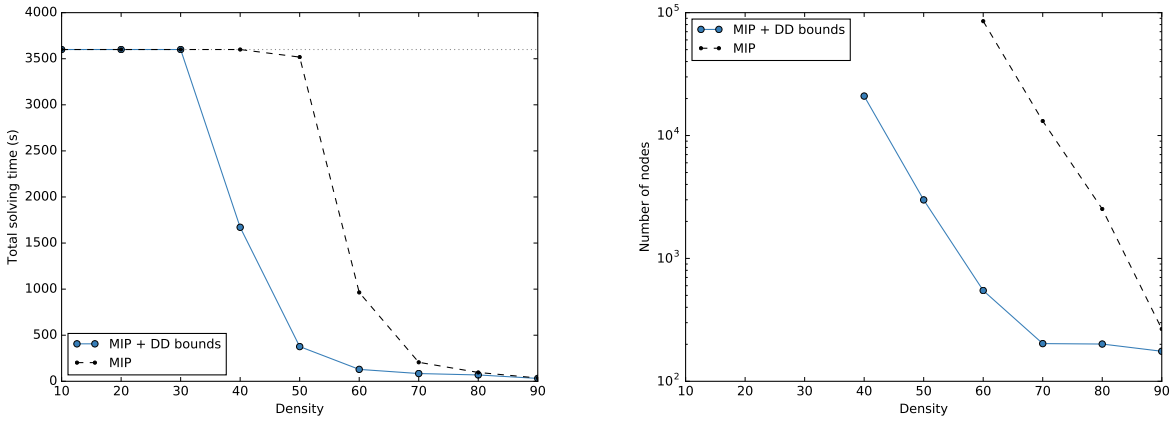


Figure 5.7: Comparison in solving time and branch-and-bound tree size between applying bounds from decision diagrams and not applying them in independent set instances with 300 vertices. The gray dotted line on the left plot indicates the time limit of one hour. Unreported data on number of nodes corresponds to cases in which the time limit of one hour was hit in the majority of runs.

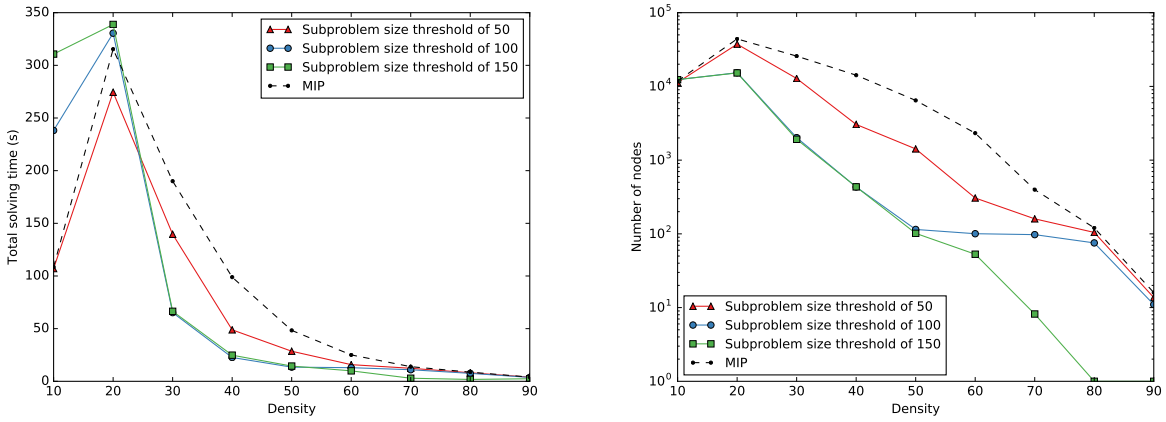


Figure 5.8: Comparison of different subproblem size thresholds with 150 vertices.

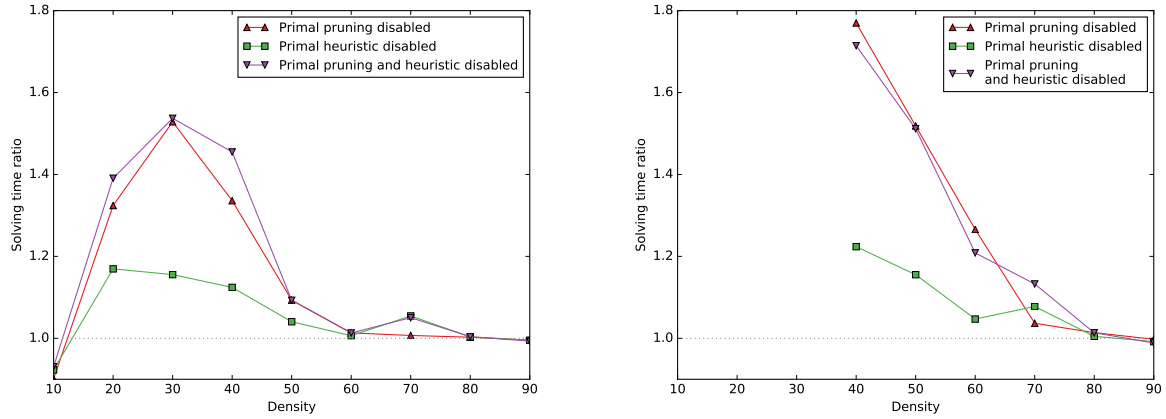


Figure 5.9: Effect on solving time when disabling primal pruning and/or primal heuristic on instances of size 150 (left) and 300 (right). More precisely, this is the ratio t_d/T, where t_d is the time without a given feature and T is the time with all capabilities. On the right, data points for densities below or equal 30 are omitted because the time limit is always hit.

		Solving time (s)		Number of nodes	
Instance	Density	MIP	MIP + DD	MIP	MIP + DD
brock200_2	50.4%	217.13	58.43	17100	298
brock200_4	34.2%	1785.94	258.30	148455	4236
C125.9	10.2%	22.65	38.21	1514	1514
gen200_p0.9_44	10.0%	22.41	41.88	646	646
gen200_p0.9_55	10.0%	2.71	2.66	1	1
gen400_p0.9_75	10.0%	605.96	857.52	2148	2148
hamming8-4	36.1%	108.44	67.31	1575	289
MANN_a27	0.1%	39.50	39.66	5598	5598
MANN_a45	0.04%	1041.03	1393.84	108239	108239
p_hat300-1	75.6%	162.32	79.51	7313	189
p_hat300-2	51.1%	1128.14	1144.55	23769	11749

Table 5.1: Effect of applying bounds on DIMACS benchmark instances that can be solved in less than one hour with SCIP 5.0.1.

The independent set constraints for the input graph G are modeled with a clique cover formulation. Above, $\mathcal{C}$ is a set of cliques that cover G. Each clique is generated by starting with a vertex with maximum degree and greedily adding vertices with maximum degree that form a clique with the current set.

The underlying graph G is a random graph following the Watts-Strogatz model [57], which has small-world properties and tend to work well with decision diagrams. Given the desired number of vertices n, the desired mean degree k (assumed even), and a probability p, construct a preliminary graph with n vertices arranged in a cycle and two vertices are adjacent if and only if they are within distance $k/2$ in the cycle. Then for each vertex i and outgoing edge (i, j), reassign j with probability p to another vertex (besides i or a neighbor of i) uniformly chosen at random.

We generate m_{knap} knapsack constraints where n is the number of variables. These constraints have coefficients a_{ij} chosen uniformly at random from 1 to 100 with a support of constant size 100. That is, we select 100 variables uniformly at random and let the coefficients of the remaining variables be zero. We maximize an objective with coefficients c_j also randomly chosen from 1 to 100.

Given that we have evaluated different parameters of number of knapsack constraints m_{knap} and right-hand side b_i in Chapter 3, we fix m_{knap} to $0.1n$ and b_i to 150 in all instances in this section. We vary the number of variables n from 300 to 450 in increments of 50.

For the relaxed decision diagrams, we use a width of 100. The subproblem size threshold is 100, determined by manual tuning.

We follow the framework described in Chapter 3. In particular, we use Lagrangian relaxation and constraint propagation as described in Section 3.4. Moreover, we include the primal pruning

	Instance size			
	300	350	400	450
Speed-up (%)	57.33	62.90	60.14	60.17
Node reduction (%)	73.93	67.44	63.63	57.75

Table 5.2: Speed-up and node reduction from using decision diagram bounds. Speed-up is the ratio of original solving time to the solving time with the bounds, minus one (e.g. a speed-up of 100% means twice as fast).

and the primal heuristic (Lagrangian relaxation-based) techniques described in Section 5.4.1.

Overall performance. The overall performance of the decision diagram bounds is presented in Figure 5.10, along with a summary in relative terms in Table 5.2. On average for these instances, this technique results in an overall speed-up of 59.08% (or equivalently, a slowdown of 37.14% if we disable the bounds). The number of nodes is reduced by 65.49% – to almost one-third of its original size. From Table 5.2, we observe that the speed-up scales well up to the sizes we tested.

Figure 5.11 illustrates all individual instances and it suggests that the approach is fairly robust for these instances.

Lagrangian relaxation, constraint propagation, and primal bounds. Figure 5.12 shows the effect of disabling each of the techniques we use on top of the dual bound generation. While the dual bounds by themselves are strong – in part because the independent set constraints play a substantial role in defining the problem – removing from consideration either the generic constraints or the primal techniques has a significant effect on the speed-up. In particular, although removing one of the two generic constraint techniques does not affect the solving time too much, disabling both of them has a large impact.

Effect of width. We observed that changing to a decision diagram width of 1000 instead of 100 results in very similar behavior. In the instances of sizes 300, 350, and 400, the average total number of nodes remained the same, and in the 450 ones, it decreased slightly. A likely reason for this behavior is that we only focus on small subproblems that do not require large widths to be effectively tackled.

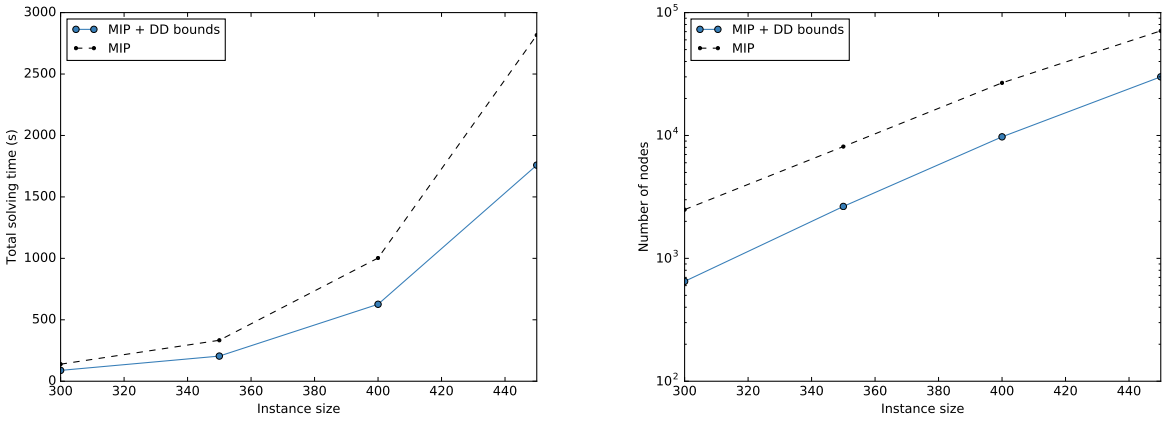


Figure 5.10: Comparison in solving time and branch-and-bound tree size of the use of decision diagram bounds for independent set + knapsack instances.

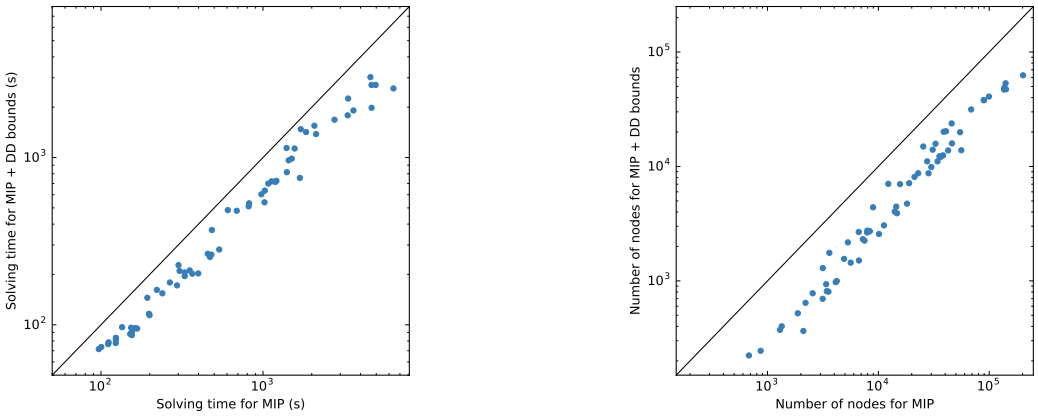


Figure 5.11: Effect of using decision diagram bounds illustrated by individual independent set + knapsack instance. Each point is an instance and points below the diagonal line correspond to better performance than not using bounds.

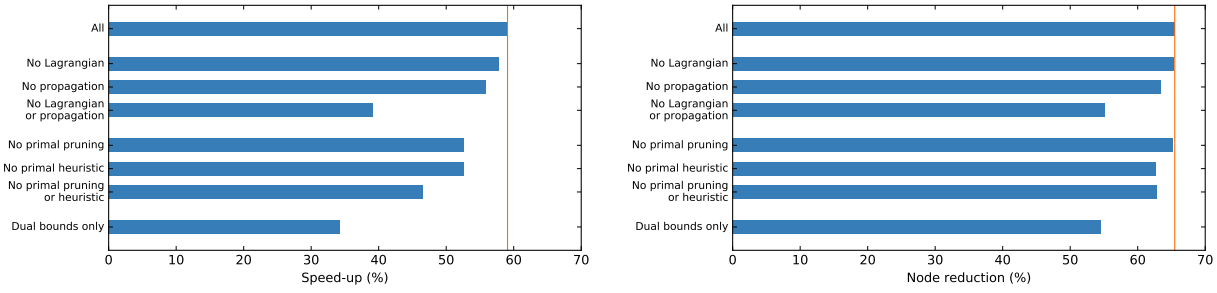


Figure 5.12: Effect of removing from consideration generic constraints or primal bounds for the independent set + knapsack instances.

5.5 Conclusion

In this chapter, we mainly explore the effectiveness of decision diagrams when used throughout the branch-and-bound tree. Between the options of generating cutting planes or bounds, we find that bounds are generally more effective throughout the tree. Part of the reason is the overhead of the target cut approach from Chapter 4, as otherwise the cuts are able to reduce the size of the branch-and-bound tree as well.

We find that the bounds are effective both in a case where the entire problem can be modeled as a decision diagram and in a case where we exploit a substructure. They are able to substantially reduce the tree size in the instances we tested, leading to a significant improvement in total solving time.

Conclusion

Relaxed decision diagrams are a powerful tool for discrete optimization. In this dissertation, we use relaxed decision diagrams to exploit substructures of integer programming models in order to improve their solution processes. The flexibility of decision diagrams enables us to develop several ways that these relaxations can be used in the context of integer programming, including to preprocess models, to generate cutting planes, to provide primal feasible solutions, and to generate dual bounds for the branch-and-bound tree. The computational experiments in this dissertation indicate that relaxed decision diagrams can effectively aid integer programming solvers for instances where structure is present.

Decision diagrams are ultimately used in this dissertation as vessels to store discrete relaxations, whether by focusing on substructures or by merging states within construction, and thus their computational performance is tied to the strength of these relaxations. Further investigating how to obtain strong decision diagram relaxations for different classes of linear constraints is valuable to expand the applicability of these techniques for a wider range of integer linear programming models.

Moreover, these techniques can be extended beyond linear constraints in inequality-based models. In integer linear programming, the linear relaxation augmented with cutting planes may already be reasonably tight in several cases. We believe that the methodology developed in this dissertation is particularly well suited when the continuous relaxation is not very strong, as we are in a sense bypassing it. In fact, it has been shown that cuts from decision diagrams in the context of nonlinear optimization (which succeeds our work on cuts) can be strong [34]. In general, discrete problems lacking good continuous relaxations are promising candidates to benefit from decision diagrams.

Beyond inequality-based models, the literature on decision diagrams for optimization continues to grow. A remarkable property of decision diagrams is their flexibility: they can be used for several different purposes and it is often smooth to transfer DD-based techniques from one context to the other. In particular, any progress on creating stronger relaxations can be used in conjunction with the techniques presented in this dissertation. There is still a lot of research to be done in this area and we expect decision diagrams to become more and more prominent in the discrete optimization literature as the overall methodology evolves.

Bibliography

[1] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. "A new look at BDDs for pseudo-boolean constraints". In: *Journal of Artificial Intelligence Research* 45 (2012), pp. 443–480.

[2] T. Achterberg. "Conflict analysis in mixed integer programming". In: *Discrete Optimization* 4.1 (2007), pp. 4–20.

[3] T. Achterberg. "Constraint integer programming". PhD thesis. Technische Universität Berlin, 2009.

[4] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. "Presolve reductions in mixed integer programming". In: *ZIB Report* (2016), pp. 16–44.

[5] S. B. Akers. "Binary decision diagrams". In: *IEEE Transactions on Computers* 100.6 (1978), pp. 509–516.

[6] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. "A constraint store based on multivalued decision diagrams". In: *Principles and Practice of Constraint Programming–CP 2007*. Springer, 2007, pp. 118–132.

[7] K. Andersen and Y. Pochet. "Coefficient strengthening: a tool for reformulating mixed-integer programs". In: *Mathematical programming* 122.1 (2010), p. 121.

[8] M. F. Anjos, F. Liers, G. Pardella, and A. Schmutzer. "Engineering branch-and-cut algorithms for the equicut problem". In: *Discrete Geometry and Optimization*. Springer, 2013, pp. 17–32.

[9] B. Aspvall, M. F. Plass, and R. E. Tarjan. "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Information Processing Letters* 8.3 (1979), pp. 121–123.

[10] A. Atamtürk, G. L. Nemhauser, and M. W. Savelsbergh. "Conflict graphs in solving integer programming problems". In: *European Journal of Operational Research* 121.1 (2000), pp. 40–55.

[11] B. Becker, M. Behle, F. Eisenbrand, and R. Wimmer. "BDDs in a branch and cut framework". In: *Experimental and Efficient Algorithms*. Springer, 2005, pp. 452–463.

[12] M. Behle. “Binary decision diagrams and integer programming”. PhD thesis. Saarbrücken, Germany: Max Planck Institute for Computer Science, 2007.

[13] D. Bergman and A. A. Cire. “Discrete nonlinear optimization by state-space decompositions”. In: *Management Science* (2017).

[14] D. Bergman and A. A. Cire. “Multiobjective optimization by decision diagrams”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 86–95.

[15] D. Bergman, A. A. Cire, and W.-J. van Hoeve. “MDD propagation for sequence constraints”. In: *Journal of Artificial Intelligence Research* 50 (2014), pp. 697–722.

[16] D. Bergman, A. A. Cire, and W.-J. van Hoeve. “Lagrangian bounds from decision diagrams”. In: *Constraints* 20.3 (2015), pp. 346–361.

[17] D. Bergman, W.-J. van Hoeve, and J. N. Hooker. “Manipulating MDD relaxations for combinatorial optimization”. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2011, pp. 20–35.

[18] D. Bergman, A. A. Cire, W.-J. van Hoeve, and T. Yunes. “BDD-based heuristics for binary optimization”. In: *Journal of Heuristics* 20.2 (2014), pp. 211–234.

[19] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. Hooker. *Decision diagrams for optimization*. Springer, 2016.

[20] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. “Discrete optimization with decision diagrams”. In: *INFORMS Journal on Computing* 28.1 (2016), pp. 47–66.

[21] D. Bergman, M. Bodur, C. Cardonha, and A. A. Cire. “Network models for multiobjective discrete optimization”. In: *arXiv preprint arXiv:1802.08637* (2018).

[22] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. “Optimization bounds from binary decision diagrams”. In: *INFORMS Journal on Computing* 26.2 (2013), pp. 253–268.

[23] D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. “Variable ordering for the application of BDDs to the maximum independent set problem”. In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2012, pp. 34–49.

[24] T. Bonato, M. Jünger, G. Reinelt, and G. Rinaldi. “Lifting and separation procedures for the cut polytope”. In: *Mathematical Programming* 146.1-2 (2014), pp. 351–378.

[25] E. A. Boyd. “On the convergence of Fenchel cutting planes in mixed-integer programming”. In: *SIAM Journal on Optimization* 5.2 (1995), pp. 421–435.

[26] R. E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.

[27] C. Buchheim, F. Liers, and M. Oswald. "Local cuts revisited". In: *Operations Research Letters* 36.4 (2008), pp. 430–433.

[28] C. Buchheim, F. Liers, and M. Oswald. "Speeding up IP-based algorithms for constrained quadratic 0–1 optimization". In: *Mathematical Programming* 124.1-2 (2010), pp. 513–535.

[29] C. Buchheim, F. Liers, and L. Sanità. "An exact algorithm for robust network design". In: *Network Optimization*. Springer, 2011, pp. 7–17.

[30] F. Cadoux and C. Lemaréchal. "Reflections on generating (disjunctive) cuts". In: *EURO Journal on Computational Optimization* 1.1-2 (2013), pp. 51–69.

[31] V. Chvátal, W. Cook, and D. Espinoza. "Local cuts for mixed-integer programming". In: *Mathematical Programming Computation* 5.2 (2013), pp. 171–200.

[32] A. A. Cire and W.-J. van Hoeve. "Multivalued decision diagrams for sequencing problems". In: *Operations Research* 61.6 (2013), pp. 1411–1428.

[33] M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer programming*. Springer, 2014.

[34] D. Davarnia and W.-J. van Hoeve. "Outer approximation for integer nonlinear programs via decision diagrams". Submitted.

[35] *DIMACS maximum clique benchmark set*. URL: http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark.

[36] M. L. Fisher. "An applications oriented guide to Lagrangian relaxation". In: *Interfaces* 15.2 (1985), pp. 10–21.

[37] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. "Approximate compilation of constraints into multivalued decision diagrams". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2008, pp. 448–462.

[38] J.-B. Hiriart-Urruty and C. Lemaréchal. *Fundamentals of convex analysis*. Springer, 2001.

[39] S. Hoda, W.-J. Van Hoeve, and J. N. Hooker. "A systematic approach to MDD-based constraint programming". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2010, pp. 266–280.

[40] K. Hosaka, Y. Takenaga, T Kaneda, and S. Yajima. "Size of ordered binary decision diagrams representing threshold functions". In: *Theoretical Computer Science* 180.1-2 (1997), pp. 47–60.

[41] R. G. Jeroslow. "On defining sets of vertices of the hypercube by linear inequalities". In: *Discrete Mathematics* 11.2 (1975), pp. 119–124.

[42] B. Kell. "Decision diagrams for combinatorial optimization and satisfaction". PhD thesis. 2015.

[43] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. 2004.

[44] J. Kinable, A. A. Cire, and W.-J. van Hoeve. "Hybrid optimization methods for time-dependent sequencing problems". In: *European Journal of Operational Research* 259.3 (2017), pp. 887–897.

[45] D. E. Knuth. *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[46] Y.-T. Lai, M. Pedram, and S. B. Vrudhula. "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.8 (1994), pp. 959–975.

[47] C.-Y. Lee. "Representation of switching circuits by binary-decision programs". In: *Bell System Technical Journal* 38.4 (1959), pp. 985–999.

[48] R. K. Martin, R. L. Rardin, and B. A. Campbell. "Polyhedral characterization of discrete dynamic programming". In: *Operations Research* 38.1 (1990), pp. 127–138.

[49] S.-i. Minato. "Zero-suppressed BDDs for set manipulation in combinatorial problems". In: *30th Conference on Design Automation*. IEEE. 1993, pp. 272–277.

[50] G. Perez. "Decision Diagrams: Constraints and Algorithms". Chapter 6. PhD thesis. Université Cote D'Azur, 2017.

[51] M. W. Savelsbergh. "Preprocessing and probing techniques for mixed integer programming problems". In: *ORSA Journal on Computing* 6.4 (1994), pp. 445–454.

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

[53] M. Sellmann. "Cost-based filtering for shorter path constraints". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2003, pp. 694–708.

[54] T. Serra and J. N. Hooker. "Compact representation of near-optimal integer programming solutions". Submitted.

[55] R. Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[56] C. Tjandraatmadja and W.-J. van Hoeve. "Incorporating bounds from decision diagrams into MIP solving". In: *Mixed-Integer Programming Workshop 2017 (Poster)*. 2017.

[57] D. J. Watts and S. H. Strogatz. "Collective dynamics of 'small-world' networks". In: *Nature* 393.6684 (1998), p. 440.

[58] I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. Vol. 4. SIAM, 2000.