Carnegie Mellon University

MELLON COLLEGE OF SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy
in Algorithms, Combinatorics and Optimization

TITLE Decision Diagrams for Combinatorial Optimization and Satisfaction

PRESENTED BY Brian Kell

ACCEPTED BY THE DEPARTMENT OF Mathematical Sciences

Willem-Jan van Hoeve MAJOR PROFESSOR	May 2015 DATE
Thomas Bohman DEPARTMENT HEAD	May 2015 DATE

APPROVED BY THE COLLEGE COUNCIL

Frederick J. Gilman DEAN	May 2015 DATE

Carnegie Mellon University
Department of Mathematical Sciences

Doctoral Dissertation

Decision Diagrams for Combinatorial Optimization and Satisfaction

Brian Kell

May 2015

Submitted to the Department of Mathematical Sciences
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Algorithms, Combinatorics, and Optimization

Dissertation Committee

Willem-Jan van Hoeve, chair
James Cummings
Alan Frieze
Ashish Sabharwal

Abstract

In this thesis we develop techniques for applying binary decision diagrams (BDDs) and multivalued decision diagrams (MDDs) to combinatorial optimization and satisfaction problems, in particular multidimensional bin packing problems and the Boolean satisfiability problem.

In the multidimensional bin packing problem, each bin has a multidimensional capacity and each item has an associated multidimensional size. We develop several MDD representations for this problem and explore different MDD construction methods including a new heuristic-driven depth-first compilation scheme. We also derive MDD restrictions and relaxations, using a novel application of a clustering algorithm to identify approximate equivalence classes among MDD nodes. Our experimental results show that these techniques can significantly outperform solvers using current constraint programming and mixed-integer programming methods.

The Boolean satisfiability (SAT) problem is the problem of determining whether a given propositional formula defined on a set of Boolean variables has a satisfying assignment, that is, an assignment of truth values to the variables that makes the formula true. We present a generic method for deducing valid clauses from a SAT instance using BDDs, with the aim of finding clauses that are bottlenecks for SAT solvers using conflict-directed clause learning. We formally characterize the strength of these generated clauses and show that any clause learned from SAT conflict analysis can also be generated using our method, while our method can additionally generate stronger clauses than those that are derivable using one application of conflict analysis. The method remains valid for approximate BDDs, which is important for SAT instances that are too large for an exact BDD representation. Our experimental results show that the clauses generated from BDDs can significantly reduce the numbers of decisions and conflicts encountered by a SAT solver.

In order to extend these clause generation techniques to larger SAT instances, we propose several methods to decompose a SAT instance into

smaller subinstances. These methods are based on the identification of clauses that arise from the application of the widely used Tseitin transformation, the analysis of the structure of the constraint graph corresponding to the instance, and the modeling of the instance as a resistive electrical network. Preliminary experiments demonstrate that these techniques are promising areas for future research.

Contents

Acknowledgements

I am deeply grateful to my advisor, Willem-Jan van Hoeve, for his continuous direction, encouragement, and advice over the past several years. This thesis could not have been completed without his guidance.

It has also been a great pleasure to work with Ashish Sabharwal of the Allen Institute for Artificial Intelligence. His knowledge and insight have been invaluable, and he has made many helpful suggestions for my work and this thesis.

I also wish to thank James Cummings and Alan Frieze for their participation on my dissertation committee, for teaching reading and topics courses in combinatorics and decision diagrams, and for maintaining a continual interest in my research work.

Further thanks go to the wonderful staff of the Department of Mathematical Sciences, Stella Andreoletti, Ferna Hartman, P. J. McCarthy, Jeff Moreci, and Nancy Watson, for their efficient and friendly help with just about everything. My gratitude is also due to Russell C. Walker and Deborah Brandon, who have been extremely supportive in my teaching and have provided many opportunities to me.

During my time at Carnegie Mellon, I have been fortunate to meet and work with an excellent group of graduate students. In particular, William Gunther, who has put up with me as an officemate since the beginning, and Marla Slusky, my academic sister, have been great companions as we all finished up this past year, and I wish them both the best of success as we begin the next chapter of our lives. I also thank Deepak Bal, Patrick Bennett, Will Boney, Jacob Davis, Lisa Espig, Jenny Iglesias, Chris Lambie-Hanson, Misha Lavrov, Emily McGregor, Paul McKenney, Clive Newstead, Jason Rute, Brendan Sullivan, Andy Zucker, and all of my other fellow graduate students.

Finally, I must thank my parents, Tom and Patti Kell, for their unceasing love and support. Mom and Dad, you have always been there for me, and I dedicate this thesis to you.

Chapter 1

Introduction

In this thesis we investigate decision diagrams, which are compact graphical representations of sets of assignments of values to variables. Our aim is to develop effective techniques to apply decision diagrams to combinatorial optimization and satisfaction problems, in particular a multidimensional bin packing problem and the Boolean satisfiability problem.

These two problems are both well known to be NP-complete [31, 39], and consequently it is widely believed that no polynomial-time algorithms exist to solve them. However, instances of these problems arise in practice [e.g., 9, 25, 41, 55, 63, 67, 68, 75, 87, 91], so there is a need to find effective methods to solve instances of practical interest. Much existing work has been done to apply techniques from constraint programming [82, 84, 86, 89], linear and mixed-integer programming [6, 7, 24, 91], and clause learning via resolution proofs [8, 71] to these problems. The algorithms and modeling techniques that we describe in this thesis improve upon the performance of these existing methods, as demonstrated by experiment.

We begin by describing new variants of algorithms to construct decision diagrams that use auxiliary state information for each node in order to enable a top-down compilation. This is not the conventional way to construct and use decision diagrams, but the top-down construction method and the use of state information for nodes are particularly well suited to searching for a feasible solution to a satisfaction problem and to deriving witnesses of infeasibility for particular nodes. The use of state information also allows the construction of approximate decision diagrams, which is useful when an exact decision-diagram representation would be too large.

We then specialize and expand upon these techniques in order to apply them effectively to the multidimensional bin packing problem and the Bool-

ean satisfiability problem. In particular, we show that a technique based on decision diagrams can significantly outperform current constraint programming and mixed-integer programming methods on the multidimensional bin packing problem, and that decision diagrams can be used to deduce valid clauses from instances of the Boolean satisfiability problem that are stronger than the clauses derivable with one application of the clause learning techniques most commonly used by modern solvers.

Finally, we explore the problem of decomposing a Boolean satisfiability instance into smaller subinstances. This is an important problem because many such instances are too large to work with effectively as a whole, even with approximate decision diagrams. We describe an algorithm to detect subinstances that correspond to propositional formulas converted to conjunctive normal form by the widely used Tseitin transformation. We also describe decomposition methods based on the graph structure of an instance, in particular a novel technique that models a Boolean satisfiability instance as a resistive electrical network.

1.1 Preliminaries

A *constraint satisfaction problem* (CSP) is specified by a set of constraints $\{C_1, \ldots, C_p\}$ on a set of variables $\{x_1, \ldots, x_n\}$ having domains $D_1, \ldots, D_n$, respectively. A *solution* to a CSP is an n-tuple $(y_1, \ldots, y_n) \in D_1 \times \cdots \times D_n$. A solution is *feasible* if the set of assignments $x_1 = y_1, \ldots, x_n = y_n$ satisfies every constraint C_j. Note that, by this usage, a "solution" is not necessarily feasible; a solution is merely a full assignment of values to variables.

A *decision diagram* is an edge-labeled acyclic directed multigraph whose nodes are arranged in $n+1$ layers $L_1, \ldots, L_{n+1}$. The layer L_1 consists of a single node, called the *root*. Every edge in the decision diagram is directed from a node in L_i to a node in L_{i+1}. All of the edges directed out of a node have distinct labels. The nodes in layer L_{n+1} are called *sinks* or *terminals*. Some of the decision diagrams we discuss in this thesis (in particular, those in Chapter 4) may also have sinks in higher layers. A node that is not a sink is called a *branch node*.

A *binary decision diagram* (BDD) is a decision diagram whose edge labels are Boolean values (0 and 1, representing false and true, respectively). A *multivalued decision diagram* (MDD) is a decision diagram whose edge labels are taken from an arbitrary (finite) set.

Commonly BDDs have two sinks: one of them, labeled $\top$, is called the *true sink* and represents satisfaction, while the other, labeled $\bot$, is called

the *false sink* and represents falsification. On the other hand, the MDDs we discuss will have a single sink (representing satisfaction), but the ideas can easily be generalized to MDDs with multiple sinks [98].

A BDD represents a Boolean function, that is, a Boolean-valued function f on the Boolean variables $x_1, \ldots, x_n$. The layers $L_1, \ldots, L_n$ correspond respectively to the variables $x_1, \ldots, x_n$; we say that a (non-sink) node in layer L_i *branches* on the variable x_i. Note that this property implies that the BDD is *ordered*, meaning that every path from the root to a sink passes through nodes that branch on the variables $x_1, \ldots, x_n$ in a sequence that respects the same linear ordering. A path from the root to a sink corresponds to values of these variables; a "true" edge from a node in layer L_i to a node in layer L_{i+1} corresponds to $x_i = 1$, while a "false" edge corresponds to $x_i = 0$. If the path corresponding to the values of $x_1, \ldots, x_n$ ends at the true sink, then $f(x_1, \ldots, x_n) = 1$; otherwise the path ends at the false sink, and $f(x_1, \ldots, x_n) = 0$. Naturally a BDD can also be viewed as representing a set of assignments of values to the variables $x_1, \ldots, x_n$, that is, a subset of $\{0, 1\}^n$: it represents the set of such assignments for which the corresponding path from the root ends at the true sink. Therefore, in particular, a BDD can be used to represent the set of assignments of values to variables that make a particular propositional formula true.

For example, consider the majority function on three variables x_1, x_2, and x_3, defined by $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. This formula is true if and only if at least two of the three variables are true. A BDD representing the set of satisfying assignments of values to these variables is shown in Figure 1.1. All edges point downward; solid lines represent "true" edges, and dashed lines represent "false" edges. The labels along the left side show the variables that are being branched on in each layer. Note that every path from the root at the top to the true sink at the bottom represents an assignment of values to the variables that makes the majority function true.

Let I be an instance of a CSP. Similarly to the way a BDD represents a set of assignments of values to Boolean variables, an MDD M with one sink representing satisfaction can be used to represent a set of solutions to I [4]. Again, the layers $L_1, \ldots, L_n$ correspond respectively to the variables $x_1, \ldots, x_n$ in I. An edge directed from a node in L_i to a node in L_{i+1} and having the label y_i, where $y_i \in D_i$, corresponds to the assignment $x_i = y_i$. Therefore a path from the root to the sink along edges labeled $y_1, \ldots, y_n$ corresponds to the solution $(y_1, \ldots, y_n)$. The MDD M represents the set $\mathscr{M}$ of solutions corresponding to all such paths.

In general, an MDD representing the set of feasible solutions to an instance of a CSP may be of exponential size, so it is useful to be able to

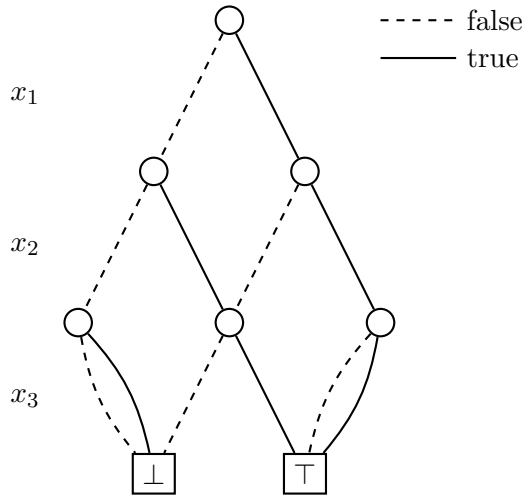

Figure 1.1: BDD for the majority function on three variables.

approximate the set of feasible solutions with a smaller MDD. Let $\mathscr{F}$ denote the set of feasible solutions to I. If $\mathscr{M} = \mathscr{F}$, $\mathscr{M} \supseteq \mathscr{F}$, or $\mathscr{M} \subseteq \mathscr{F}$, then M is said to be an *exact MDD*, a *relaxation MDD*, or a *restriction MDD* for I, respectively [4, 11, 14, 45]. Collectively, relaxation and restriction MDDs are called *approximate MDDs*, because the sets of solutions they represent are approximations of $\mathscr{F}$. Note that exact MDDs are both relaxation MDDs and restriction MDDs.

Likewise we have relaxation BDDs and restriction BDDs. Let F be a propositional formula on the variables $x_1, \ldots, x_n$, and let $\mathscr{F}$ be the set of assignments of values to these variables that satisfy F (so $\mathscr{F} \subseteq \{0,1\}^n$). Let B be a BDD that represents the subset $\mathscr{B} \subseteq \{0,1\}^n$. Then B is an exact BDD, a relaxation BDD, or a restriction BDD, accordingly, if $\mathscr{B} = \mathscr{F}$, $\mathscr{B} \supseteq \mathscr{F}$, or $\mathscr{B} \subseteq \mathscr{F}$.

It should be noted that the definition of decision diagrams we have given here differs in two important ways from the conventional definition.

First, the usual definition of decision diagrams allows edges to "skip" layers: an edge may go from a node in layer L_i to a node in layer L_{i+k} for $k \geq 1$. (These decision diagrams are normally still required to be ordered.) An edge that skips layers can be interpreted in several ways. One common interpretation is that the variables whose corresponding layers are skipped may be assigned any value. Another interpretation, followed by so-called *zero-suppressed BDDs* (ZDDs or ZBDDs) [60, 98], is that the variables whose

corresponding layers are skipped must be assigned the value zero. ZDDs are useful to represent sparse Boolean functions, that is, Boolean functions whose support (the set of assignments to the variables $x_1, \ldots, x_n$ that yield the value $\top$) consists mostly of assignments of values that are mostly zero. Sparse Boolean functions can be viewed as representations of set families; ZDDs were introduced by Minato [77, 78] for combinatorial applications. The algorithms that we describe in this thesis are designed for use with decision diagrams in which every edge is directed from a node in layer L_i to a node in layer L_{i+1}; none of the edges in our decision diagrams will skip layers.

Second, in the literature, decision diagrams are commonly required to be reduced. A path in a decision diagram from the root to a node in the layer L_i represents a *partial assignment* $y = (y_1, \ldots, y_{i-1}) \in D_1 \times \cdots \times D_{i-1}$. Let $\mathscr{F}(y)$ denote the set of *feasible completions* of this partial assignment, that is,

$$\mathscr{F}(y) = \{\, z \in D_i \times \cdots \times D_n : (y, z) \text{ is feasible} \,\}.$$

If y and y' are partial assignments with $\mathscr{F}(y) = \mathscr{F}(y')$, then we say that y and y' are *equivalent*. An (exact) decision diagram is *reduced* if every two equivalent partial assignments correspond to paths starting at the root and ending at the same node. A reduced decision diagram is the minimal-size representation of the corresponding set of assignments. Observe that in an exact decision diagram all paths from the root to a fixed node v represent equivalent partial assignments, and conversely if two partial assignments y and y' are equivalent then the paths in an exact decision diagram that correspond to y and y' can lead to the same node; if the decision diagram is reduced, then those paths *must* lead to the same node. In a reduced decision diagram, there exists no pair of nodes v and v' in the same layer L_i with identical (partial) mappings $\phi, \phi' \colon D_i \to L_{i+1}$ given by the outgoing edges, because if such a pair of nodes did exist then the nodes v and v' could be replaced by a single node. Additionally, if edges in the decision diagram are allowed to skip layers, then in a reduced decision diagram there exists no node v in layer L_i with an outgoing edge for every value $y_i \in D_i$, all of which point to the same node w, because if such a node did exist all incoming edges to v could simply be redirected to point to w, and the node v could be deleted. The decision diagrams we consider in this thesis are not necessarily reduced in this structural sense. However, they are reduced in a weaker sense: no two nodes in the same layer have the same *state* (see Section 2.1).

Furthermore, in this thesis we are neither constructing nor using deci-

sion diagrams in the conventional way. We associate semantic information with each node, called the state of the node, and this information is used for top-down construction, the identification of infeasible nodes, and the computation of bounds and witnesses of infeasibility. The conventional method for constructing BDDs uses a "bottom-up approach," called *synthesis* [22], based on a product operation called *melding* that combines BDDs for Boolean functions f and g to produce a BDD for a Boolean function $f \diamond g$, where $\diamond$ is a Boolean operation such as $\wedge$ or $\oplus$; see Knuth [60] or Wegener [98] for details. This is not the approach we take here. Rather, we construct a decision diagram in a single top-down pass, starting at the root and using the state information of the nodes to determine the structure of the next layer. Our construction algorithms are described in full in Chapter 2.

1.2 History and previous work

The work in this thesis builds upon previous work in the theory of decision diagrams and the problems of bin packing and Boolean satisfiability. We briefly survey this work in this section.

1.2.1 Decision diagrams

The groundbreaking paper of Shannon [88] in 1938 applied the theory of Boolean algebra to the design and analysis of electrical switching circuits. The expansion theorem of Boole [20] states that any function $f(x_1, \ldots, x_n)$ on the Boolean variables $x_1, \ldots, x_n$ can be expressed as

$$\big(\overline{x}_1 \wedge f(0, x_2, \ldots, x_n)\big) \vee \big(x_1 \wedge f(1, x_2, \ldots, x_n)\big).$$

Shannon applied this expansion (along with the other laws of Boolean algebra) to develop a systematic method for the construction of switching circuits to compute Boolean functions, and it is the foundational idea of BDDs. Because of Shannon's pioneering use of this theorem in electrical engineering, it is often called Shannon expansion. Some of the specific ideas of decision diagrams are present implicitly in Shannon's 1938 paper; for example, a discussion of symmetric Boolean functions and their corresponding switching circuits in Section 4 can be seen as a demonstration that the number of branch nodes in a BDD for such a function is at most $\binom{n+1}{2}$.

Shannon's paper was followed by the paper of Lee [64] in 1959, which contrasted the algebraic representation of Shannon with a new representation Lee called "binary-decision programs." A binary-decision program is a

sequence of conditional jump instructions, each of which specifies a variable and the addresses of two later instructions (or special addresses θ and I representing the outputs 0 and 1, respectively). Such a program is executed by beginning at instruction 1 and repeatedly testing the corresponding variable; if the value of the variable is 0, then execution jumps to the instruction specified by the first address, and if the value is 1 then execution jumps to the instruction specified by the second address. This is essentially a string representation of a (not necessarily ordered or reduced) BDD in which the nodes are represented by these conditional jump instructions.

The name "binary decision diagram" was coined by Akers [2] in 1978. Akers described a bottom-up construction method beginning with a truth table for a Boolean function and a top-down method based on Shannon expansion followed by reduction. He also described an algorithm for counting the number of paths from the root to a sink and a BDD-based technique for generating test cases for circuit verification, and discussed the representation and manipulation of BDDs by digital computers.

Much of the modern interest in decision diagrams was sparked by a highly influential paper published in 1986 by Bryant [22] showing that if the conditions of orderedness and reducedness are enforced then BDDs can be an efficient representation of many useful Boolean functions, the BDD representation of a given Boolean function is unique, and many operations on such functions can be efficiently computed from their BDD representations. The synthesis algorithm was first presented in this paper.

Since Bryant's paper, a great variety of applications and generalizations of decision diagrams have been proposed and used in practice; see the books by Knuth [60], Meinel and Theobald [76], and Wegener [98] for examples and further information.

In 2007, Behle [10] investigated *threshold BDDs*, which are BDDs representing the set of 0–1 solutions $x = (x_1, \ldots, x_n)$ satisfying a threshold function of the form $a^{\mathrm{T}}x \leq b$ for some weight vector a. The size of such a BDD is bounded by a polynomial in n, provided that the weights are polynomially bounded. A threshold inequality like this is very closely related to the knapsack problem. Behle presented a top-down algorithm for constructing the BDD for a given threshold inequality, and introduced a parallel AND operation for combining multiple BDDs simultaneously. If the size of the final BDD is small, this parallel operation can avoid an explosion in the size of the intermediate BDDs that may occur if pairwise AND operations were performed sequentially. Behle also investigated the optimal variable ordering problem, which asks for the ordering of the variables $x_1, \ldots, x_n$ that minimizes the size of the BDD; he presented a 0–1 integer program-

ming formulation of this problem that can be used to compute the variable ordering spectrum of a threshold function.

Multivalued decision diagrams were introduced by Srinivasan et al. [92] in 1990 as a generalization of binary decision diagrams for direct application to combinatorial problems. Of particular relevance to this thesis are the recent applications of MDDs and approximate MDDs to constraint programming. Approximate MDDs of limited width and corresponding propagation algorithms were proposed by Andersen et al. [4] as a way to implement richer constraint stores for CP solvers while meeting space restrictions. This idea was developed further by Hadzic et al. [45], systematized by Hoda et al. [47], and applied by Cire and van Hoeve [28] and Bergman et al. [12] for problems involving sequencing constraints. Decision diagrams have also been used by Bergman et al. [11, 13] to obtain bounds on objective functions for many types of optimization problems, and the use of restriction BDDs to generate heuristic solutions was investigated in Bergman et al. [14].

1.2.2 Bin packing

The classical one-dimensional bin packing problem consists of a list $L = (a_1, \ldots, a_n)$ of items, each having a size in the interval $(0, 1]$, and the goal is to pack them into a minimum number of bins having capacity 1. This problem was first formalized and discussed by Eilon and Christofides [37]. Other early papers include those of Garey et al. [40], Johnson [51], and Johnson et al. [52]. The bin packing problem is a well known NP-complete problem; it appears in the extensive list of NP-complete problems of Garey and Johnson [39]. For this reason, a large amount of work has been done on approximation algorithms for the bin packing problem. The reader is referred to Coffman et al. [30] for a recent survey and classification of this approximation work.

A related problem is the *cutting stock problem*, in which a number of items of various sizes must be cut from stock that comes in units of a fixed size. The main difference between the one-dimensional bin packing problem and the one-dimensional cutting stock problem is that the items in a bin packing problem tend to have widely differing sizes, while a cutting stock problem tends to have many items of the same size. This difference, while apparently subjective, has led to different approaches for the two problems [65]. Dyckhoff [34] and Wäscher et al. [97] discuss and categorize various types of cutting and packing problems in the literature.

In 2004, Shaw [89] presented a CP constraint for bin packing. This was improved and extended in the doctoral thesis of Schaus [84] in 2009, and

was further refined and applied to a tank allocation problem in Schaus et al. [86].

The specific multidimensional variant of the bin packing problem that we consider in this thesis, which is also known as the *multidimensional vector packing* problem, was introduced in Garey et al. [41] in the context of multiprocessor scheduling with resource constraints. Recent work includes that of Spieksma [91], who presented a branch-and-bound algorithm; Caprara and Toth [24], who investigated lower bounds and combinatorial and integer programming formulations; and Bansal et al. [6, 7], who used randomized rounding of the linear programming relaxation. Further applications of the problem were investigated by Beck and Siewiorek [9], who modeled the problem of task allocation for embedded, bus-based multicomputers and examined heuristic solution techniques; Chang et al. [25], who modeled the efficient packing of steel coils into containers for shipping; and Shachnai and Tamir [87], who modeled the problem of data placement on disks in media-on-demand systems.

1.2.3 Boolean satisfiability

The Boolean satisfiability (SAT) problem is the archetypal NP-complete problem, as shown in the celebrated paper of Cook [31]. It is the problem of determining whether a given propositional formula defined on a set of Boolean variables has a satisfying assignment, that is, an assignment of truth values to the variables that makes the formula true.

The SAT problem arises frequently in practical applications, including hardware verification [15, 16, 75, 90, 96], software model checking and testing [29, 50, 58], automatic test-pattern generation [61, 63, 70, 93], combinational equivalence checking [21, 62, 69], planning in artificial intelligence [55, 83], and scheduling [42]. Specific applications of SAT techniques have been used for crosstalk noise prediction in integrated circuits [26], the solution of open problems in group theory [99], termination analysis in term-rewrite systems [38], and haplotype inference in bioinformatics [67]. For a survey and discussion of some applications of Boolean satisfiability, the reader is referred to Marques-Silva [68].

Many advances have been made in the development of SAT solvers in recent decades, and SAT solvers can now be used to solve large-scale instances involving millions of variables and constraints. Much of the success of modern SAT solvers stems from their ability to quickly learn new constraints from infeasible search states via conflict-directed clause learning (CDCL). This technique of conflict analysis and clause learning was first proposed by

Marques-Silva and Sakallah [72, 73] in the late 1990s and implemented in a solver called GRASP. In 2001, Moskewicz et al. [79] presented a solver called Chaff that was able to solve many unsatisfiable instances two orders of magnitude faster than previous solvers; it introduced several improvements to the technique used by GRASP, including new heuristics, a restart strategy, and an efficient implementation of Boolean constraint propagation based on "watched literals." For an overview of the application of CDCL to SAT solving, see Marques-Silva et al. [71]. Conflict analysis has also been applied in the context of mixed-integer programming [1, 59] and constraint programming [32, 53, 80, 94] as "nogood" learning.

A formal characterization of the strength of clause learning techniques was given by Beame et al. [8]. Katsirelos et al. [54] found that the solution of SAT instances using CDCL often involves bottlenecks: there exist single clauses that the solver must deduce in order to proceed and upon which all future progress depends, but which themselves require significant work to discover.

1.3 Contributions and outline

We make the following contributions in this thesis.

In Chapter 2, we describe several techniques for the construction of decision diagrams. Algorithm 1, described in Section 2.1, is a new generalization of a top-down construction algorithm first described in Bergman et al. [11]; our algorithm allows for exploratory construction if the purpose of the construction is to seek a feasible solution to a satisfiability problem, as described in Section 2.2. In Section 2.3, we give a variant of another algorithm of Bergman et al. for the construction of approximate decision diagrams, that is, decision diagrams that represent the solution sets of relaxations or restrictions of problem instances. Our algorithm uses a novel application of the median cut algorithm of Heckbert [46] to determine sets of nodes to be merged.

As a demonstration of these algorithms, we apply them to the solution of a multidimensional bin packing problem in Chapter 3. We discuss two representations of instances of this problem as MDDs in Sections 3.2 and 3.3; the second representation, called the ullage MDD representation, is better able to exploit symmetry in the problem instance. In Section 3.4, we make these ideas practicable by describing precisely how the ullage MDD representation fits into the model used by the construction algorithms of Chapter 2 and examining some techniques to improve the performance of the algorithms

by identifying and eliminating redundant computations. Our experimental results, reported in Section 3.5, show that our approach can provide significant performance improvements over current constraint programming and mixed-integer programming methods.

The focus of Chapter 4 is the use of BDDs to deduce valid clauses from instances of the Boolean satisfiability problem with the aim of improving the performance of existing solvers. After a discussion of the BDD representation of SAT instances in Section 4.1, we explore three new methods for deducing clauses from BDDs in Section 4.2. The last of these methods, which generates a witness clause for each infeasible node in the BDD as proof of its infeasibility, is examined more thoroughly in Section 4.3. Here we formally characterize the clauses deduced in this way and show that any clause learned from SAT conflict analysis, the standard clause-deduction mechanism in modern SAT solvers, can also be generated using our method, while our method can additionally generate stronger clauses than those that can be derived from one application of conflict analysis. This method remains valid for approximate BDDs, so it can be applied in practice for instances that are too large for an exact BDD representation. Chapter 4 concludes with a discussion of experimental results, reported in Section 4.4, which show that the clauses deduced using our method can significantly reduce the numbers of conflicts and decisions encountered by a SAT solver.

The practical implementation of the techniques described in Chapters 3 and 4 requires certain considerations to be made, and we discuss some of these design choices in Chapter 5.

In order to extend the applicability of our clause-generation method of Chapter 4 to larger SAT instances, it is desirable to be able to decompose a SAT instance into smaller subinstances and work with the subinstances instead of the instance as a whole. This is the focus of Chapter 6. In Section 6.1, we present a well-known method, the Tseitin transformation, to convert an arbitrary propositional formula into conjunctive normal form, and then describe a novel algorithm to do the reverse, i.e., to detect subsets of clauses in a SAT instance that were generated by the Tseitin transformation. In Section 6.2, we describe ways in which a SAT instance can be represented by a graph and thereby decomposed into subinstances. This graph representation is used in a different way in Section 6.3, in which the graph is interpreted as a resistive electrical network. We present a new algorithm, using this electrical model, to determine "neighborhoods" of clauses in a SAT instance and thereby decompose the instance into subsets of similar clauses.

We conclude in Chapter 7 with a summary of the thesis and proposals

for future investigation.

Portions of the work in this thesis have been previously published as Kell and van Hoeve [56] and Kell et al. [57].

Chapter 2

Construction of decision diagrams

In this chapter we discuss algorithms for the construction of an MDD representing the set of feasible solutions to a CSP. In particular, we present a generic exploratory construction algorithm for MDDs and an application of the median cut algorithm of Heckbert [46] in the construction of approximate MDDs. The content of this chapter is joint work with Willem-Jan van Hoeve [56].

2.1 Exact decision diagram construction

The standard algorithm in the literature for constructing BDDs is called *synthesis* [22]. At the core of this algorithm is a product operation called *melding* that combines BDDs for Boolean functions f and g to produce a BDD for a Boolean function $f \diamond g$, where $\diamond$ is a Boolean operation such as $\wedge$ or $\oplus$; see Knuth [60] or Wegener [98] for details. The synthesis algorithm is a "bottom-up" approach to BDD construction.

We describe and use a very different approach in this thesis. The algorithms that we present in this chapter provide "top-down" construction methods for decision diagrams. A key part of our methods is the association of auxiliary state information with each node. This state information allows us to identify equivalent portions of the decision diagram as it is being constructed. In Chapter 4 we show that the state information can be useful even after the construction process is complete.

Recall from Section 1.1 that a path in an MDD from the root to a node in the layer L_i represents a partial assignment $y = (y_1, \ldots, y_{i-1}) \in$

$D_1 \times \cdots \times D_{i-1}$. Let $\mathscr{F}(y)$ denote the set of feasible completions of this partial assignment, that is,

$$\mathscr{F}(y) = \{\, z \in D_i \times \cdots \times D_n : (y, z) \text{ is feasible} \,\}.$$

If y and y' are partial assignments with $\mathscr{F}(y) = \mathscr{F}(y')$, then we say that y and y' are *equivalent*. Because equivalent partial assignments have the same set of feasible completions, the recognition that two partial assignments are equivalent reduces the size of the MDD, because the corresponding paths can lead to the same node.

In general, determining whether two partial assignments are equivalent is an NP-hard problem (because it is NP-hard even to determine whether a partial assignment for a CSP has a feasible completion). However, we can sometimes determine that two partial assignments are equivalent by associating partial assignments with "states." A *state function* for the layer L_i is a map σ_i from the set $Y_i = D_1 \times \cdots \times D_{i-1}$ of partial assignments at layer L_i to some set S_i of *states*, such that $\sigma_i(y) = \sigma_i(y')$ implies $\mathscr{F}(y) = \mathscr{F}(y')$. In other words, two partial assignments that lead to the same state have the same set of feasible completions. (A "perfect" state function would also allow us to say that two partial assignments that lead to different states have different sets of feasible completions, and we strive for this ideal, but for practical reasons our state function should be easy to compute, so we cannot require this.)

The notion of states of partial assignments has been used in previous work. The top-down algorithm described in Akers [2] constructs a BDD for a Boolean function by using Shannon expansions directly, recognizing equivalent partial assignments by the fact that the corresponding subfunctions are syntactically identical. Behle [10] described a top-down algorithm for the construction of threshold BDDs, which are exact representations of solution sets of instances of 0–1 knapsack problems; his work implicitly uses states of partial assignments. A general algorithm for a top-down, layer-by-layer (i.e., breadth-first) construction of an MDD is presented as Algorithm 1, "Top-down MDD compilation," in Bergman et al. [11]. The key to the top-down construction of an MDD is the identification of a *node equivalence test*, which determines when two nodes on the same layer (each representing one or more partial assignments) have the same set of feasible completions; this is exactly what a state function does.

So far we have spoken of the states of partial assignments. We shall now extend this idea to states of nodes in an MDD. In the MDD that we construct, partial assignments to the variables $x_1, \ldots, x_{i-1}$ that lead to the

same state will correspond to paths from the root that lead to the same node in layer L_i; we shall associate this state with this node. Now, given a node v in layer L_i in the MDD and its state, which we shall write as state(v), and given a value $y_i \in D_i$, we can determine the state of a child node w of v if the edge (v, w) has label y_i. This is simply the state of the partial assignment (y, y_i), where y is any partial assignment corresponding to the node v.

Example 2.1.1. Consider a CSP on three variables x_1, x_2, and x_3, having domains $D_1 = \{0, 1, 2\}$, $D_2 = \{1, 3\}$, and $D_3 = \{0, 1\}$, respectively, with the single constraint $x_1 + x_2 + x_3 \geq 3$.

Given a partial assignment of values to the variables, the important information about that partial assignment is the sum of the assigned values. From this information alone, we can determine whether or not a completion of the partial assignment (i.e., an assignment of values to the rest of the variables) is feasible. So a reasonable choice for the state of a partial solution is this sum. Of course, once the sum reaches 3, we do not need to keep track of its exact value—every completion of the partial assignment will be feasible. Therefore, the set of states can be $\{0, 1, 2, \star\}$, where $\star$ represents a sum that is greater than or equal to 3.

The MDD in Figure 2.1 represents the set of feasible solutions to this CSP. (Note that this MDD has only one sink, representing satisfaction.) Any two partial assignments whose corresponding paths lead to the same node in this MDD have the same state (i.e., the same sum of assigned values), so we associate that state with the node itself. The node states are shown in the figure as labels.

Suppose we are given the state of a node v in layer L_i in this MDD and a value $y_i \in D_i$. Let w be the node in layer L_{i+1} to which the outgoing edge of v labeled y_i points. Then we can determine state(w) as follows: if state(v) $= \star$ or state(v) $+ y_i \geq 3$, then state(w) $= \star$; otherwise state(w) $=$ state(v) $+ y_i$. ∎

To be more precise, and to make these ideas applicable to generic CSPs, we make the following definitions. Let $i \in \{1, \ldots, n+1\}$. Let $Y_i = D_1 \times \cdots \times D_{i-1}$ denote the set of partial assignments corresponding to paths from the root to a node in layer i; take $Y_1 = \{\emptyset\}$, a singleton set having one element representing the empty partial assignment. Let S_i be an arbitrary set whose elements are called *states* and which contains a special element $\perp$ indicating infeasibility. Recall that we say that $\sigma_i : Y_i \to S_i$ is a *state function* if $\sigma_i(y) = \sigma_i(y')$ implies $\mathscr{F}(y) = \mathscr{F}(y')$; we also require that $\sigma_i(y) = \perp$ implies $\mathscr{F}(y) = \emptyset$. We assume that we can test the feasibility of a (complete)

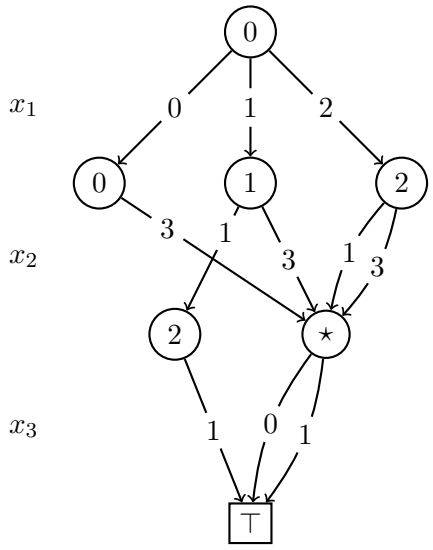

Figure 2.1: MDD for the CSP $x_1 \in \{0,1,2\}$, $x_2 \in \{1,3\}$, $x_3 \in \{0,1\}$, $x_1 + x_2 + x_3 \geq 3$. The node labels are states.

assignment, so for $y \in Y_{n+1}$ we require that $\sigma_{n+1}(y) = \bot$ if $\mathscr{F}(y) = \emptyset$. For $i \in \{1,\ldots,n\}$, we say that $\chi_i : S_i \times D_i \to S_{i+1}$ is a *child state function* if $\chi_i\big(\sigma_i(y), y_i\big) = \sigma_{i+1}(y, y_i)$ for all $y \in Y_i$ and all $y_i \in D_i$.

In order to use state information effectively in the construction of an MDD, we must maintain, for each layer L_i, a mapping from states to nodes that have already been constructed in L_i. When we seek a node in L_i having state s, we consult this mapping to see if such a node already exists. Such a mapping can be implemented with a hash table. It is often called the *unique table* because it ensures that the node representing state s in layer L_i is unique [60].

Algorithm 1 constructs an exact MDD. For each $i \in \{1,\ldots,n+1\}$ let σ_i be a state function, and for each $i \in \{1,\ldots,n\}$ let χ_i be a corresponding child state function. Let r be the root node. The algorithm maintains a collection T of nodes to be processed, i.e., nodes whose children need to be constructed. When a node v in layer i is processed, each possible domain value $y \in D_i$ is considered, and the corresponding child state s is computed. If the child state is not obviously infeasible (i.e., if it is not $\bot$), then the unique table is consulted to see if a node w with state s already exists in layer L_{i+1}. If no such node already exists, a new node w is constructed in layer L_{i+1} and added to T. Then the edge (v,w) is added to the MDD with

label y. On the other hand, if the child state is infeasible, an edge from v to a false sink can be constructed if desired (this is useful for the algorithms in Chapter 4, for example). These steps are repeated until all nodes have been processed.

Algorithm 1 Exact MDD construction

```
1:  L_1 := {r}
2:  T := {r}                                          ▷ nodes to be processed
3:  while T is not empty do
4:      select v ∈ T and remove it from T
5:      i := layer(v)                                 ▷ i.e., v ∈ L_i
6:      for all y ∈ D_i do
7:          s := χ_i(state(v), y)                     ▷ determine child state
8:          if s ≠ ⊥ then          ▷ child state is not obviously infeasible
9:              w := unique-table(i + 1, s)           ▷ does node with state s exist?
10:             if w = nil then
11:                 w := new node with state s
12:                 add w to L_{i+1}
13:                 add w to T
14:             end if
15:             add edge (v, w) with label y
16:         else                                      ▷ child state is infeasible
17:             if desired, add edge (v, ⊥) with label y
18:         end if
19:     end for
20: end while
```

2.2 Exploratory construction

The main difference between Algorithm 1 and the top-down exact MDD compilation algorithm of Bergman et al. [11] is the order in which the nodes are processed. Instead of requiring that the nodes be processed layer by layer, we allow the collection T to provide the nodes in any order. This generalization permits exploratory construction of the MDD. For example, if we are constructing the MDD in order to seek a feasible solution, we can build it in a depth-first manner by taking T to be a stack. The layer-by-layer behavior of the algorithm of Bergman et al. can be achieved by using a queue for T. Note that if we do construct the MDD layer by layer, we can

discard the unique table for each layer as soon as we have finished processing the previous layer.

It is useful to have a heuristic to estimate the "promise" of a partial assignment. For example, we may have a heuristic to estimate the likelihood that a partial assignment has a feasible completion (for an instance of a satisfiability problem) or to estimate the optimal value of a feasible completion (for an instance of an optimization problem). Such a heuristic can be used to guide the depth-first construction of an MDD in search of a feasible solution or an optimal solution. With such a heuristic, we can use a priority queue for T to select the most promising nodes to process next. Alternatively, we can use a stack for T and modify Algorithm 1 slightly so that when we process a node we construct all its children, evaluate their heuristics, and then add them to T in reverse order of their promise. This will yield a depth-first algorithm that explores the most promising child of each node first.

This depth-first MDD construction process, especially if it is being used simply to find a feasible solution, is very similar to a backtracking search. It is an improvement, however, because the MDD nodes act as a memoization technique to prevent the exploration of portions of the search tree that can be recognized as equivalent to portions already explored.

2.3 Approximate MDDs

In general, exact MDDs can be of exponential size, so the use of Algorithm 1 may not be practical because of space limitations. In this case we may be able to use an approximate MDD to get useful results.

Recall from Section 1.1 that an approximate MDD represents a superset or a subset of the set of feasible solutions to a CSP, which is to say that an approximate MDD represents a set of solutions to a relaxation or a restriction of the problem instance. (It is important to note that the individual solutions represented by an approximate MDD are not themselves "approximate" solutions to the problem instance; rather, it is the *set* of feasible solutions that is being approximated.) Hence, if a restriction MDD indicates that an instance is feasible, then every solution it represents (i.e., every path from the root to the sink) is an exact feasible solution to the original instance. Similarly, an indication of infeasibility from a relaxation MDD is a proof that the original instance is infeasible. In this way, relaxation and restriction MDDs can be used together to determine the feasibility or infeasibility of an instance and to get an exact feasible solution if the instance

is feasible. Of course, it is possible for a relaxation MDD to indicate that an instance is feasible while a restriction MDD indicates it is infeasible, in which case nothing is learned. In response, one could construct MDDs representing tighter relaxations or restrictions (probably at the cost of greater time and space requirements) or could embed the MDDs inside a complete search.

2.3.1 Approximation MDDs by merging

MDDs of limited width were proposed by Andersen et al. [4] to reduce space requirements. In this approach, the MDD is constructed in a top-down, layer-by-layer manner; whenever a layer of the MDD exceeds some preset value W, an approximation operation is applied to reduce its size to W before constructing the next layer. For this approximation, Bergman et al. [11] use a relaxation operation $\oplus$ defined on states of nodes so that, given nodes v and v', the state given by $\text{state}(v) \oplus \text{state}(v')$ is a "relaxation" of both $\text{state}(v)$ and $\text{state}(v')$; see also Hoda et al. [47].

We can formalize this idea as follows. Let $C_i = D_i \times \cdots \times D_n$ denote the set of all possible assignments of values to the variables $x_i, \ldots, x_n$ (independent of any particular partial assignment for the variables $x_1, \ldots, x_{i-1}$). For a partial assignment $y \in Y_i$, the set of feasible completions of y is some subset of C_i, so $\mathscr{F}(y) \in \mathscr{P}(C_i)$, where $\mathscr{P}$ denotes the power set. Recall that a state function $\sigma_i : Y_i \to S_i$ is such that $\sigma_i(y) = \sigma_i(y')$ implies $\mathscr{F}(y) = \mathscr{F}(y')$, and $\sigma_i(y) = \bot$ implies $\mathscr{F}(y) = \emptyset$. The existence of such a function implies the existence of a *completion function* $\tau_i : S_i \to \mathscr{P}(C_i)$ such that $\tau_i\big(\sigma_i(y)\big) = \mathscr{F}(y)$ for all $y \in Y_i$. For $i \in \{1, \ldots, n\}$, we say that a binary operation $\vee_i : S_i \times S_i \to S_i$ is a *relaxation merge* if for all $y, y' \in Y_i$ we have $\tau_i\big(\sigma_i(y) \vee_i \sigma_i(y')\big) \supseteq \mathscr{F}(y) \cup \mathscr{F}(y')$. In other words, the set of feasible completions implied by the state $\sigma_i(y) \vee_i \sigma_i(y')$ contains all feasible completions implied by the state $\sigma_i(y)$ and all feasible completions implied by the state $\sigma_i(y')$. Similarly, we call $\wedge_i : S_i \times S_i \to S_i$ a *restriction merge* if for all $y, y' \in Y_i$ we have $\tau_i\big(\sigma_i(y) \wedge_i \sigma_i(y')\big) \subseteq \mathscr{F}(y) \cap \mathscr{F}(y')$. For simplicity, we shall omit the subscript and just write $\vee$ or $\wedge$. These merge operations need not be associative or commutative. However, in a slight abuse of notation, we shall write $\bigvee A$ to denote a combination of all elements $s \in A \subseteq S_i$ using the relaxation merge operation $\vee$, in any order and parenthesized in any way; likewise for $\bigwedge A$.

Bergman et al. [11] give an algorithm to construct a limited-width MDD which iteratively merges pairs of nodes in a layer using a relaxation merge. We propose a refinement of this technique that uses a clustering algorithm

to partition the nodes in the layer into W clusters; the nodes in each cluster are then merged into a single node.

The framework algorithm for top-down approximate MDD construction is given in Algorithm 2. Line 20 in this algorithm calls a subroutine to reduce the size of the layer L_{i+1} to W when necessary. This can be a call to Algorithm 3, which reduces the size of a layer by merging.

Algorithm 2 Top-down approximate MDD construction

```
 1: L_1 := {r}
 2: for i = 1 to n do
 3:     L_{i+1} := ∅
 4:     for all v ∈ L_i do
 5:         for all y ∈ D_i do
 6:             s := χ_i(state(v), y)                    ▷ determine child state
 7:             if s ≠ ⊥ then          ▷ child state is not obviously infeasible
 8:                 w := unique-table(i + 1, s)
 9:                 if w = nil then
10:                     w := new node with state s
11:                     add w to L_{i+1}
12:                 end if
13:                 add edge (v, w) with label y
14:             else                                     ▷ child state is infeasible
15:                 if desired, add edge (v, ⊥) with label y
16:             end if
17:         end for
18:     end for
19:     if |L_{i+1}| > W then                                  ▷ layer too large
20:         reduce the size of L_{i+1} to W
21:     end if
22: end for
```

To perform the clustering of nodes on line 1 of Algorithm 3, we adapted the median cut algorithm of Heckbert [46], which was originally designed for color quantization of images. The median cut algorithm operates on a set of points in q-dimensional Euclidean space (in the original version, $q = 3$, representing the red, green, and blue components of each pixel in the image) and partitions the points into clusters. Initially all of the points are grouped into a single cluster, which is tightly enclosed by a q-dimensional rectangular box. Then the following operation is repeatedly performed: the box having the longest length (among all boxes in all q dimensions) is selected, and it is

Algorithm 3 Reduction of layer L_{i+1} to size W by merging

```
1: partition L_{i+1} into W clusters A_1, ..., A_W
2: for j = 1 to W do
3:     w_j := new node with state ⋁ A_j (or ⋀ A_j)
4:     for all v ∈ A_j do
5:         change every edge (u, v) to (u, w_j) with the same label
6:     end for
7: end for
8: L_{i+1} := {w_1, ..., w_W}
```

divided into two boxes along this longest length at the median point, that is, in such a way that each of the two smaller boxes contains approximately half of the points in the original box; the two smaller boxes are then "shrunk" to fit tightly around the points they contain. This process continues until the desired number of clusters (boxes) have been generated. The median cut algorithm can be implemented to run in $O\big(K(pq + \log K)\big)$ time, where K is the desired number of clusters, p is the number of points, and q is the number of dimensions.

In order to apply the median cut algorithm to the nodes in a layer of an MDD, we interpret the state of each node as a point in q-dimensional Euclidean space, for some value of q. In order for the median cut algorithm to yield a meaningful clustering of the nodes in a layer, this interpretation should be chosen so that similar states correspond to points that are close to each other with respect to the maximum norm $\|\cdot\|_\infty$, defined by $\|(x_1, \ldots, x_q)\|_\infty = \max\{|x_1|, \ldots, |x_q|\}$.

If a merged MDD reports that a CSP is feasible, it is desirable to extract a (possible) feasible solution from it. One way to do this is to maintain a representative partial assignment for each node as the MDD is constructed; when two nodes are merged, either of the two corresponding partial assignments can be selected (perhaps in accordance with a heuristic) as the representative partial assignment for the merged node. Then the representative (complete) assignment at the sink will be a (possibly) feasible solution for the CSP. The representative partial assignment can be viewed as auxiliary state information of the node.

2.3.2 Restriction MDDs by deletion

Algorithm 2 can be used with Algorithm 3 to construct a limited-width MDD by merging nodes when the size of a layer becomes too large. If we

are constructing a restriction MDD, however, then another option is simply to delete some of the nodes in the layer [14]. The selection of nodes to keep can be guided by a heuristic. This is described in Algorithm 4.

Algorithm 4 Reduction of layer L_{i+1} to size W by deletion

1: use heuristic to select most promising nodes $w_1, \ldots, w_W \in L_{i+1}$
2: **for all** $w \in L_{i+1} \setminus \{w_1, \ldots, w_W\}$ **do**
3: delete w from L_{i+1} and delete all edges (u, w)
4: **end for**

We note that this deletion algorithm does not use a partitioning algorithm to cluster the nodes in each layer as the merging algorithm does; instead it incurs the cost of computing a heuristic for each node. So the deletion algorithm may be especially beneficial if partitioning the nodes in a layer of the MDD is slower than computing a heuristic for a node.

2.4 Summary

In this chapter we presented algorithms for the construction of decision diagrams. Our algorithms are based on the association of semantic information (states) with nodes. We described a generic top-down construction algorithm for MDDs in Section 2.1, and in Section 2.2 we described how this algorithm can be used to search for a feasible solution to a CSP by enabling heuristic-driven exploratory construction. In Section 2.3 we gave algorithms for the construction of approximate decision diagrams by merging or deletion, including an application of a clustering algorithm to determine subsets of nodes to be merged.

Chapter 3

MDDs for bin packing

In the previous chapter we presented generic MDD construction algorithms, suitable for any constraint satisfaction problem. In this chapter we specialize some of these techniques to a multidimensional bin packing problem. Our experimental results show that such techniques can yield an improvement on existing methods.

The content of this chapter is joint work with Willem-Jan van Hoeve [56].

3.1 The multidimensional bin packing problem

Many related problems in combinatorial optimization are collectively referred to as "bin packing problems." In the classical bin packing problem, the input is a list $(s_1, \ldots, s_n)$ of item sizes, each in the interval $(0, 1]$, and the objective is to pack the n items into a minimum number of bins of capacity 1.

Here we study a multidimensional variant of the bin packing problem, presented as a satisfaction problem. An instance of this problem consists of a list $(s_1, \ldots, s_n)$ of item sizes and a list $(c_1, \ldots, c_m)$ of bin capacities, Each item size and each bin capacity is a d-tuple of nonnegative integers; for example, $s_i = (s_{i,1}, \ldots, s_{i,d})$. The objective is to assign each of the n items to one of the m bins in such a way that, for every bin and in every dimension, the total size of the items assigned to the bin does not exceed the bin capacity.

This can be viewed as a CSP with n variables and md constraints. Each variable x_i has domain $\{1, \ldots, m\}$ and denotes the bin into which the ith item is placed. The constraints require that $\sum_{i:x_i=j} s_{i,k} \leq c_{j,k}$ for all $j \in \{1, \ldots, m\}$ and all $k \in \{1, \ldots, d\}$.

Note that the "dimensions" in this problem should not be interpreted as geometric dimensions. In this way the problem studied here differs from the two- and three-dimensional bin packing problems studied, for example, by Lodi et al. [66] and by Martello et al. [74], in which the items and bins are geometric rectangles or cuboids. Rather, the dimensions in the problem studied here correspond to independent one-dimensional bin packing constraints that must be satisfied simultaneously.

Multidimensional bin packing problems of the kind considered here (also known as multidimensional vector packing problems) appear in practice, especially to model resource allocation problems. For example, Garey et al. [41] introduced the problem in the context of multiprocessor scheduling with resource constraints; Beck and Siewiorek [9] have modeled the problem of task allocation for embedded, bus-based multicomputers; Chang et al. [25] have modeled the efficient packing of steel coils into containers for shipping; Shachnai and Tamir [87] have modeled the problem of data placement on disks in media-on-demand systems; and the ROADEF/EURO Challenge 2012[1] involved a set of machines with several resources, such as RAM and CPU, running processes which consume those resources. However, in comparison to other variants of bin packing, this particular multidimensional variant has received relatively little attention in the literature. Current CP methods are weak on problems involving simultaneous bin packing constraints. Current MIP methods do better but are still limited in their effectiveness.

3.2 Direct MDD representation

Let I be a multidimensional bin packing instance, having n items and m bins. A direct MDD representation of the set of feasible solutions of I has layers $L_1, \ldots, L_n$ corresponding to the variables $x_1, \ldots, x_n$ and also the last layer L_{n+1} which contains the sink. The edge labels are elements of $\{1, \ldots, m\}$. A path from the root to the sink along edges labeled $y_1, \ldots, y_n$ represents the feasible solution $(y_1, \ldots, y_n)$, that is, the feasible solution in which item i is placed into bin y_i.

After items having sizes $s_{i_1}, \ldots, s_{i_k}$ have been placed into a bin of capacity c_j, the remaining capacity of the bin is $c_j - \sum_{l=1}^{k} s_{i_l}$. (Recall that the item sizes and bin capacities are d-tuples; here and elsewhere in this chapter addition and subtraction of d-tuples is done componentwise.) We

[1] Société française de Recherche Opérationnelle et Aide à la Décision. ROADEF/EURO Challenge 2012: Machine Reassignment. `http://challenge.roadef.org/2012/en/`.

shall call this remaining capacity the *ullage* of the bin; it is a d-tuple. (The word "ullage" means "the amount by which a container falls short of being full.") Of course, the ullage of each bin is nonincreasing (componentwise) as the items are placed one by one into the bins.

A useful state function for the direct MDD representation is the map σ_i from a partial solution $y = (y_1, \ldots, y_{i-1})$ to the list $(u_1, \ldots, u_m)$ of the ullages u_j of the m bins; in other words, for $j \in \{1, \ldots, m\}$, we take

$$u_j = c_j - \sum_{k \in K_j} s_k,$$

where $K_j = \{\, 1 \le k \le i-1 : y_k = j \,\}$.

As an example, Figure 3.1 shows the direct MDD representation for a one-dimensional bin packing instance having two bins, each of capacity 7, and four items, with sizes 5, 3, 2, and 1. There are six paths from the root to the sink, representing the six feasible solutions; for instance, the path following the edges labeled 2, 1, 1, 2 corresponds to the solution in which the item of size 5 is packed in bin 2, the items of size 3 and 2 are packed in bin 1, and the item of size 1 is packed in bin 2. The node labels are the states. For instance, the path from the root along the edges labeled 1, 2, 2 represents a partial solution for which the ullages of the two bins are each 2, so the state of this partial solution is $(2, 2)$. The partial solution corresponding to the path 2, 1, 1 has the same state. Observe that if two partial solutions at layer i have the same lists of ullages, then they have the same set of feasible completions, so this is indeed a state function.

If exploratory construction is desirable, as described in Section 2.2, then it is useful to have a heuristic to estimate the "promise" of a partial solution, that is, the likelihood that it has a feasible completion. For the multidimensional bin packing problem, we propose the following heuristic. Given a partial solution $(y_1, \ldots, y_i)$ describing the packing of the first i items into bins, we perform a non-backtracking random packing of the remaining items $(i+1, \ldots, n)$ as well as we can without violating the bin packing constraints. In other words, we iterate through the remaining items in order, and we pack each item into one of the bins that has sufficient ullage, chosen at random; if no such bin exists, we put the item into a trash pile. At the end we count the total size of the items in the trash pile, along all d dimensions, and this number is the score of this packing. This random packing of the remaining items is repeated several times, and the total score of these packings is used as the heuristic value of the partial solution; a low score is better. (Occasionally, while we are computing the heuristic for a partial solution

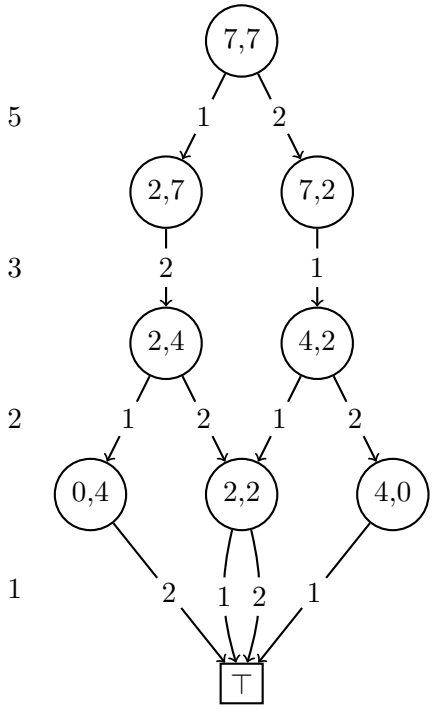

Figure 3.1: Direct MDD representation of a multidimensional bin packing instance. The node labels are the states, and the numbers to the left of the MDD are the sizes of the items being packed in each layer.

in this way, we may luckily find a feasible completion: the trash pile will be empty. In this case, if we are constructing the MDD merely to seek a feasible solution, we can immediately return the solution thus found.)

The construction of an approximate MDD by merging requires an appropriate merge operation, as described in Section 2.3.1. For the direct MDD representation of a multidimensional bin packing instance, node states are lists of ullages $(u_1, \ldots, u_m)$, so an appropriate relaxation merge is the componentwise maximum and an appropriate restriction merge is the componentwise minimum.

Also, the use of the median cut algorithm for clustering requires the interpretation of the state of each node as a point in q-dimensional Euclidean space, for some value of q. For the direct MDD representation, the state of a node is a list of d-dimensional ullages, one for each of the m bins; so we view this state directly as an md-dimensional point.

3.3 Ullage MDD representation

Let I be a multidimensional bin packing instance, having n items and m bins. One difficulty with the direct MDD representation of I is that it does not take into account the possible symmetry of the bins. For example, suppose that item 1 will fit in any of the m bins. Then the root of the direct MDD will have m outgoing edges labeled 1 through m, indicating the possible bins into which item 1 can be packed. However, if the bins are all identical, these possibilities are essentially equivalent (up to a reordering of the bins). The direct MDD representation cannot recognize this equivalence, because the sets of feasible completions, corresponding to edge-labeled paths in the MDD, are different. For example, in Figure 3.1, the two edges directed out of the root node represent essentially equivalent choices.

To address the possible symmetry of the bins, we can reduce the number of distinct descriptions of feasible solutions by expressing the solutions differently. Rather than assigning items directly to bins, we assign each item to an ullage. For example, instead of saying that item 3 is packed into bin 2, we say that it is packed into a bin with ullage 4. We call this the *ullage description* of the solution; it consists of a list $(u_1, \ldots, u_n)$ of d-tuples, assigning an ullage to each item.

To specify the domains of the variables u_i in the ullage description of a solution, we define the *ullage multiset function* U. If $C = (c_1, \ldots, c_m)$ is the list of bin capacities in I, then $U\big(C, (u_1, \ldots, u_i)\big)$ denotes the multiset of ullages after the first i items have been placed into bins as described by

the list $(u_1, \ldots, u_i)$. This is the same as the multiset of ullages after the first $i-1$ items have been placed, except that an item of size s_i was placed into a bin having ullage u_i; so an element u_i of the multiset should be removed and replaced with an element $u_i - s_i$. Formally, we can define U recursively as follows:

- $U(C, \emptyset) = C$ (viewing C as a multiset).
- For $i \in \{1, \ldots, n\}$, if $U_{i-1} = U\big(C, (u_1, \ldots, u_{i-1})\big)$ is defined and $u_i \in U_{i-1}$, then $U\big(C, (u_1, \ldots, u_i)\big) = (U_{i-1} \setminus u_i) \cup \{u_i - s_i\}$.

With this definition of U, the domain of the variable u_i in the ullage description of a solution is $U\big(C, (u_1, \ldots, u_{i-1})\big)$. Note that this domain depends on the values that have previously been assigned to $u_1, \ldots, u_{i-1}$.

An *ullage MDD representation* of the set of feasible solutions of I has layers $L_1, \ldots, L_{n+1}$. The label of an edge directed out of a node in layer L_i in an ullage MDD is a d-tuple, representing the ullage of the bin into which item i is to be placed (after items 1 through $i-1$ have been placed into bins). Therefore the edge labels $u_1, \ldots, u_n$ along a path from the root to the sink in an ullage MDD correspond to an ullage description $(u_1, \ldots, u_n)$ of a feasible solution to I.

Figure 3.2 illustrates the ullage MDD representation for the same one-dimensional bin packing instance as earlier, having two bins of capacity 7 and items with sizes 5, 3, 2, and 1. At the root, the state is $\{7 \times 2\}$, i.e., a multiset containing the element 7 with multiplicity 2. The first item, of size 5, must be placed in a bin having ullage 7; this leads to the state $\{2, 7\}$. Then the second item, of size 3, must be placed in the bin that now has ullage 7, and so forth. Of course, a path from the root to the sink in this ullage MDD can easily be converted into an explicit list of bin assignments if desired.

3.4 State function for the ullage MDD representation

For the ullage MDD representation, it is useful to consider the state of a partial assignment having ullage description $(u_1, \ldots, u_{i-1})$ to be the multiset of ullages of the bins, that is, $U\big(C, (u_1, \ldots, u_{i-1})\big)$.

This idea can be extended to handle side constraints in the CSP. For example, the steel mill slab problem [85] is essentially a (one-dimensional) bin packing problem with the additional constraint that each item has a

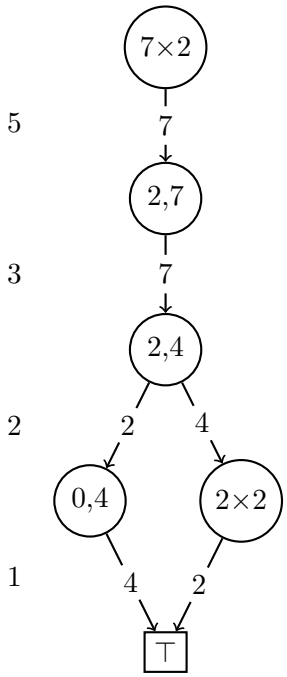

Figure 3.2: Ullage MDD representation for a multidimensional bin packing instance.

color and no bin can contain items of more than two colors. To handle a side constraint like this, we can simply augment the state information of a node to include the colors of items that have been packed into it so far.

A few observations can be used to identify additional equivalent partial assignments. Let $u_{j,k}$ denote the ullage of bin j, in the kth dimension, after we have placed items 1 through i into bins. Let a denote the greatest possible sum of a subset of the sizes of items $i+1$ through n, in the kth dimension, that does not exceed $u_{j,k}$. If $a < u_{j,k}$, then we may consider the ullage of bin j, in the kth dimension, to be a rather than $u_{j,k}$ without changing the set of feasible completions. Using this technique of "rounding down" the ullages across all bins in all dimensions, we can sometimes identify additional equivalent partial assignments (their states may be the same after they are rounded down, even if they were not the same before). Moreover, after rounding down ullages, we may discover that the total ullage in all bins is not enough for the remaining items; then we know that the current state has no feasible completions.

If, after we have placed items 1 through i into bins, there is any bin that is so small that none of the remaining items will fit, we can declare that bin *dead* and remove it from further consideration. This is potentially stronger than rounding down, because it may be that in each dimension, considered separately, there is some remaining item that will fit into the bin; but no remaining item is small enough in every dimension to fit into the bin.

Conversely, if after we have placed items 1 through i into bins, there is some bin that is large enough in every dimension that all of the remaining items will fit in it, then we know that the instance is feasible. We call such a bin *free*. Once we discover a free bin, we can immediately return a feasible solution: extend a partial assignment corresponding to the current node to a complete assignment by packing all remaining items into the free bin.

The ideas underlying the concepts of dead and free bins are present in Behle's threshold BDD algorithm [10].

In Figure 3.3 we apply the rounding-down technique to the ullage MDD. If we additionally check for dead and free bins, we will discover a free bin in the second layer (the bin with rounded-down ullage of 6).

3.5 Experimental results

We implemented the MDD-based algorithms described above in Java, using the exploratory construction method described in Section 2.2, the approximation methods from Section 2.3, the ullage MDD representation described

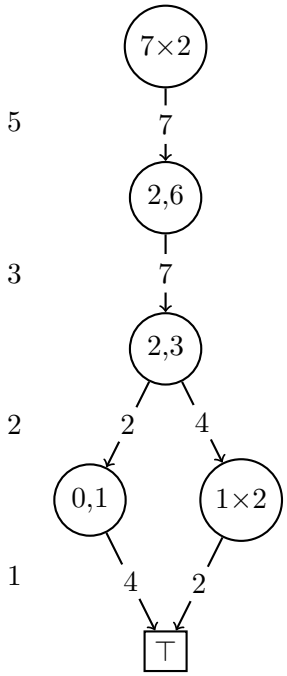

Figure 3.3: Ullage MDD representation for a multidimensional bin packing instance, with ullages rounded down.

in Section 3.3, and the state function and improvements described in Section 3.4. See Chapter 5 for additional implementation details.

Our test instances were generated as follows. Given values for the parameters d (the number of dimensions), n (the number of items), m (the number of bins), and β (percentage bin slack), we first generate a list of n item sizes $(s_1, \ldots, s_n)$, each of which is a d-tuple whose coordinates are integers chosen uniformly and independently at random from $\{0, \ldots, 1000\}$. Then the sum $t = \sum_{i=1}^{n} s_i$ is computed, and the m bin capacities are all taken to be $\lceil (1 + \beta/100)t/m \rceil$; these computations are done componentwise. (If $\beta = 20$, for example, then the total bin capacity, in each dimension, will be 20% more than the total item size.) An instance is rejected and regenerated if it contains any single item that is too large to be placed into a bin, as such an instance is obviously infeasible.

Our test instances have 6 dimensions, 18 items, and 6 bins; we generated 52 such instances for each integer value of β from 0 to 35. These instances are available at `http://www.math.cmu.edu/~bkell/6-18-6-instances.txt` or by request; the 52 instances with 20% bin slack are given in Appendix A.

By their construction, these instances have identical bins. The ullage MDD representation can exploit this symmetry effectively to reduce the number of branches in the search tree. This is especially evident in the infeasible instances, where infeasibility must be established by some kind of exhaustive search.

The experiments were run on an 32-bit Intel Pentium 4 CPU at 3.00 GHz with 1 GiB of RAM using Windows 7 Professional. The maximum Java heap size was set to 512 MiB. We used AIMMS 3.13 with CPOptimizer 12.4 as the constraint programming (CP) solver and CPLEX 12.4 as the mixed-integer programming (MIP) solver, with their default settings.

The CP model has n variables $x_1, \ldots, x_n$, each with domain $\{1, \ldots, m\}$; the assignment $x_i = j$ indicates that item i is packed into bin j. These variables are subject to d independent `cp::BinPacking` constraints [89].

The MIP model has mn binary variables $x_{i,j}$, for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$; the assignment $x_{i,j} = 1$ indicates that item i is packed into bin j. The MIP model also has a nonnegative "overflow" variable ω_j for each bin, representing the maximum amount by which the bin is overfull in any dimension, and there is a nonnegative "total overflow" variable $\Omega = \sum_{j=1}^{m} \omega_j$. The MIP model is shown in Figure 3.4. It is formulated as a minimization problem only because that is the form the solver requires; the constraint $\Omega = 0$ means it is really just a feasibility problem.

We compared the performance of CP and MIP to our MDD approaches:

$$
\begin{aligned}
&\min \Omega \\
&\text{s.t.} \sum_{j=1}^{m} x_{i,j} = 1; \\
&\quad \sum_{i=1}^{n} s_{i,k} x_{i,j} \leq c_{j,k} + \omega_j \quad \text{for all } k \in \{1, \ldots, d\},\ j \in \{1, \ldots, m\}; \\
&\quad \Omega = \sum_{j=1}^{m} \omega_j; \\
&\quad x_{i,j} \in \{0, 1\}, \quad \omega_j \geq 0, \quad \Omega = 0.
\end{aligned}
$$

Figure 3.4: MIP model for multidimensional bin packing.

the exact MDD (using depth-first, heuristic-driven exploratory construction), a relaxation MDD using the relaxation merge operation, and restriction MDDs using the restriction merge operation or deletion. All instances were run to completion using each method. The maximum width for the approximation MDDs was set to 5000 nodes. With this width, the approximation MDDs returned "feasible" or "infeasible" correctly in all instances except two: the restriction merge MDD returned "infeasible" incorrectly for one instance with 25% bin slack and one instance with 26% bin slack. The combination of the relaxation merge MDD and the deletion (restriction) MDD was enough to correctly solve all 1872 instances.

Figure 3.5 displays the feasibility and hardness profiles for these instances. The horizontal axis corresponds to the bin slack of the instances. The thick curve in the plot is the feasibility profile: it uses the left vertical axis, and shows that the percentage of instances that are feasible is 0% when the bin slack is low and 100% when the bin slack is high, with a dramatic phase transition centered around approximately 20% bin slack. The other three curves in the plot show hardness profiles for CP, MIP, and the exact MDD method. These curves use the right vertical axis and show a prominent hardness peak near the phase transition.

In the infeasible region, on instances having bin slack between about 2% and 22%, the average run time of the exact MDD method is consistently less than that of MIP and significantly less than that of CP (by over three orders of magnitude at 20% bin slack). On the other hand, in the feasible region, on instances having bin slack more than about 25%, CP and MIP both

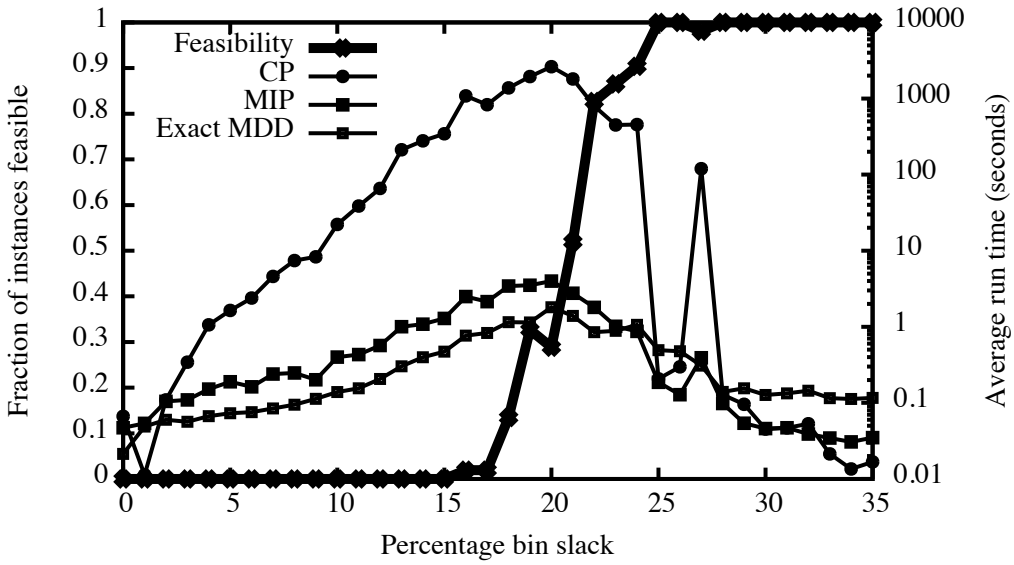

Figure 3.5: Feasibility and hardness profiles for instances having 6 dimensions, 18 items, and 6 bins.

tend to outperform the exact MDD method. A notable exception (visible as a spike in the hardness profile) occurs at 27% bin slack, for which one of the 52 generated instances happened to be infeasible; this single infeasible instance greatly increased the average run time of CP and MIP without noticeably affecting the performance of the exact MDD.

We investigated the instances at the hardness peak, i.e., those having 20% bin slack, in more detail. A performance profile for these instances appears in Figure 3.6, including CP, MIP, the exact MDD, and the combination of the relaxation merge MDD and the deletion (restriction) MDD. The CP solver required over 400 seconds for 35 instances (67%), taking almost 14,000 seconds in the extreme case. The MIP solver did much better, solving every instance in less than 12 seconds. The exact MDD method, which solved each instance in less than 6 seconds, was faster than MIP in 32 instances (62%), while the relaxation MDD and the deletion MDD together (sufficient in all 52 instances to establish feasibility or infeasibility) were faster than MIP in 24 instances (46%).

When we look only at the 37 infeasible instances with 20% bin slack, as seen in Figure 3.7, the difference between CP/MIP and the MDD approaches becomes clearer. (Restriction MDDs do not give useful results for infeasible

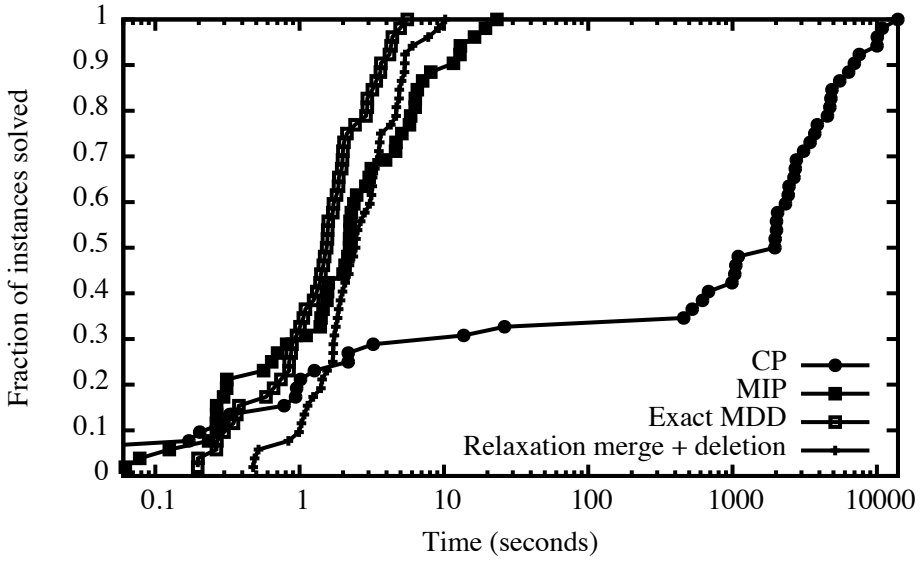

Figure 3.6: Performance profile on the subset of instances having 20% bin slack.

instances, so they are omitted from this plot merely for clarity. All of the approximate MDD methods we implemented ran about equally fast on all instances with 20% bin slack, so using a restriction MDD together with the relaxation approximately doubles the run time.)

On the other hand, in the performance profile on the 15 feasible instances with 20% bin slack, shown in Figure 3.8 (with the relaxation MDD omitted for clarity), the various methods are not so clearly separated.

We make the following observations from these experimental results. The advantage of the ullage MDD representation on infeasible instances comes from its ability to exploit the symmetry among identical bins in order to reduce the number of branches taken in an exhaustive search. The number of branches in an exhaustive search is further reduced by the techniques described in Section 3.4, namely, rounding down ullages and identifying dead bins. However, on feasible instances, our Java code, which is not particularly optimized, does not find solutions as quickly as the commercial CP and MIP solvers do. The depth-first, heuristic-driven algorithm tends to solve feasible instances more quickly than the layer-by-layer approximation algorithms, but limited-width MDDs tend to be faster than exact MDDs on infeasible instances.

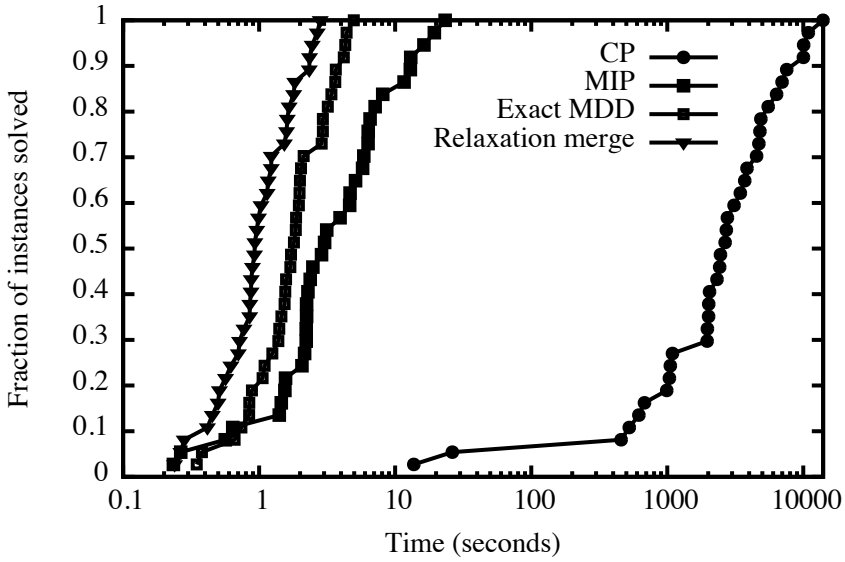

Figure 3.7: Performance profile on the infeasible instances having 20% bin slack.

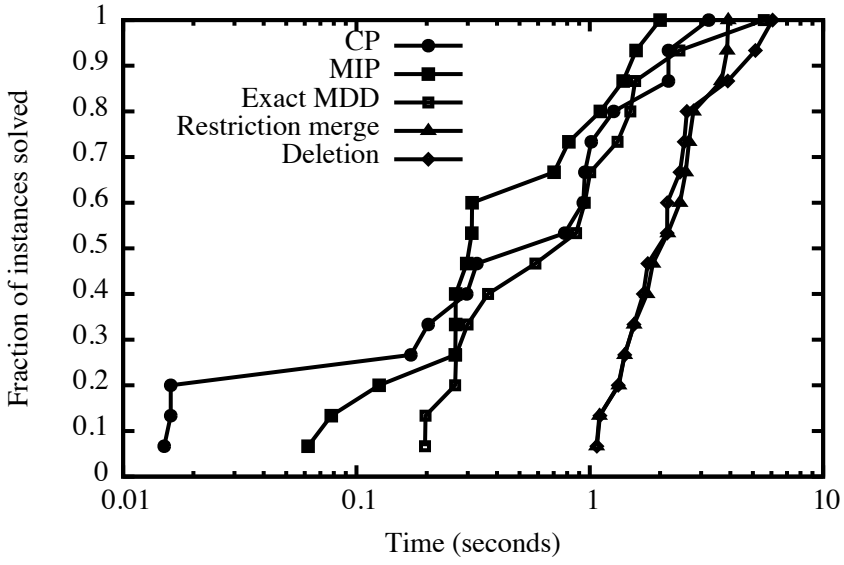

Figure 3.8: Performance profile on the feasible instances having 20% bin slack.

3.6 Summary

In this chapter we developed techniques to apply MDDs effectively to the multidimensional bin packing problem. We began by describing the problem in Section 3.1 and formulating it as a CSP. In Sections 3.2 and 3.3 we described two MDD representations for instances of the problem: the direct MDD representation and the ullage MDD representation. We showed that the ullage MDD representation can take advantage of symmetry in the instance to reduce the size of the MDD. We discussed the state function for the ullage MDD representation in greater depth in Section 3.4, including a rounding-down technique in order to detect equivalent states and the identification of free and dead bins in order to detect feasibility and infeasibility. Our experimental results, presented in Section 3.5, demonstrated that these techniques, when combined with the construction algorithms from Chapter 2, can outperform current CP and MIP solvers.

Chapter 4

BDDs for SAT clause generation

An instance of the Boolean satisfiability (SAT) problem is a propositional formula on variables $x_1, \ldots, x_n$, expressed in conjunctive normal form (CNF), that is, as a conjunction of disjunctions of *literals*, where a literal is a variable x_i or its negation $\overline{x}_i$. Each of these disjunctions is called a *clause*. Because logical conjunction and disjunction are commutative, associative, and idempotent, we may view a SAT instance as a set of clauses, each of which is a set of literals. The objective is to determine whether there exists an assignment of Boolean values to the variables that simultaneously satisfies every clause.

Many advances have been made in the development of SAT solvers in recent decades, and SAT solvers can now be used to solve large-scale instances involving millions of variables and constraints. Much of the success of modern SAT solvers stems from their ability to quickly learn new constraints from infeasible search states via conflict-directed clause learning (CDCL); for an overview of the application of CDCL to SAT solving, see the survey by Marques-Silva et al. [71]. Conflict analysis has also been applied in the context of mixed-integer programming (MIP) [1, 59] and constraint programming (CP) [53, 80, 94] as "nogood" learning. In the context of constraint programming, nogood learning techniques have been proposed for specific combinatorial structures that arise from global constraints. For example, Downing et al. [32] study nogoods for global constraints that can be represented as a network flow. However, it remains a challenge to learn effective nogoods for MIP and CP solvers in a more generic context.

In this chapter we introduce a generic approach for learning nogoods

from BDDs, both exact and approximate. We specifically focus on clause learning in the context of SAT solving, which is perhaps the most general form of nogood learning.

The architecture of today's SAT solvers, combining unit propagation with rapid restarts and CDCL, focuses on techniques with very low overhead and maximizes the number of search nodes that can be processed per second. While this has clearly been beneficial, the unit propagation inference performed by SAT solvers is arguably limited in strength. We therefore investigate a way to generate clauses that are stronger than those currently derived from unit propagation and CDCL. We show that these clauses, when added to the original formula, can substantially reduce the search tree size.

Our approach is based on a BDD representation of SAT instances. As in Chapter 3, we associate a state with each node of the BDD: the set of clauses that are not yet satisfied. This allows us to apply the top-down construction algorithms of Chapter 2, using the sets of unsatisfied clauses to determine node equivalence.

The key observation in our work is that the BDD node states, for those nodes that do not lead to a satisfying solution, can also be used to generate new clauses witnessing the infeasibility of these nodes. Such clauses can be viewed as "nogoods" that forbid the solver to visit the associated search states. Since a node in a BDD can represent multiple partial assignments, a single nogood generated in this way is as strong as multiple nogoods derived from these separate partial assignments.

In Section 4.3, we formally characterize the strength of the clauses generated by our method. For example, we show that our clauses can indeed be stronger than those generated by one invocation of traditional conflict analysis. We also show the equivalence of our approach to regular and ordered resolution, which are specific restricted forms of resolution proofs. Because exact BDD representations of SAT instances are generally not practicable, we demonstrate that our method still efficiently produces valid clauses from approximate BDDs.

In Section 4.4, we report results of computational experiments performed to evaluate the strength of our generated clauses in practice. We show that, for certain problem classes, our clauses can considerably reduce the size of the search tree. However, the solving time is not always reduced accordingly; we attribute this behavior to the length and number of our generated clauses. Nonetheless, the qualitative strength of our clauses demonstrates a great potential for inclusion in SAT solvers.

The content of this chapter is joint work with Ashish Sabharwal and Willem-Jan van Hoeve. Much of it appears in [57].

4.1 BDD representation of SAT instances

In order to apply the top-down BDD construction algorithm from Chapter 2 to construct a BDD from a SAT instance, we define $\sigma_i(y)$ for a partial assignment $y = \{y_1, \ldots, y_{i-1}\}$ to be the set of clauses in the instance that are not satisfied by the assignments $x_1 = y_1, \ldots, x_{i-1} = y_{i-1}$. Observe that if two partial assignments at layer i have the same set of unsatisfied clauses, then they have the same set of feasible completions, so this is indeed a state function.

The state of the root node is the full set of clauses in the instance (excluding tautological clauses like $x_1 \vee \overline{x}_1$, if such clauses appear), and the state of a child node is formed from the state of its parent by removing all clauses that are satisfied by the variable assignment corresponding to the edge from the parent to the child. Let $\text{vars}(C)$ denote the set of variables that appear in the clause C. If the state of a child node in layer L_i would contain a clause C such that $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$, then the clause C has not yet been satisfied by assignments to the variables $x_1, \ldots, x_{i-1}$, but it contains no variable corresponding to a lower layer of the BDD, and so C cannot be satisfied by future assignments to variables. Therefore, in this case the child state function should return $\bot$, and the edge from the parent node should point directly to the false sink.

Example 4.1.1. Consider a graph coloring problem on a complete graph with three vertices. Vertices 1 and 2 can be colored 0 or 1, while vertex 3 can be colored 0, 1, or 2. All nodes must be colored differently. We introduce variable x_1 for vertex 1, where $\overline{x}_1$ represents color 0 and x_1 represents color 1. Likewise we introduce x_2 for vertex 2. For vertex 3, we introduce three variables x_3, x_4, and x_5 for colors 0, 1, and 2, respectively. Here a positive literal represents that we choose that color, while its negation represents that we do not choose that color (e.g., $\overline{x}_3$ means that vertex 3 is not colored 0). We can formulate this problem as the following SAT instance with 11 clauses:

(1)	$x_3 \vee x_4 \vee x_5$	(7)	$\overline{x}_1 \vee \overline{x}_2$
(2)	$\overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_5$	(8)	$x_1 \vee x_4 \vee x_5$
(3)	$\overline{x}_3 \vee \overline{x}_4$	(9)	$\overline{x}_1 \vee x_3 \vee x_5$
(4)	$\overline{x}_3 \vee \overline{x}_5$	(10)	$x_2 \vee x_4 \vee x_5$
(5)	$\overline{x}_4 \vee \overline{x}_5$	(11)	$\overline{x}_2 \vee x_3 \vee x_5$
(6)	$x_1 \vee x_2$		

The constructed BDD, using the lexicographic variable ordering, is presented in Figure 4.1. The state of each node is the set of (indices of) clauses

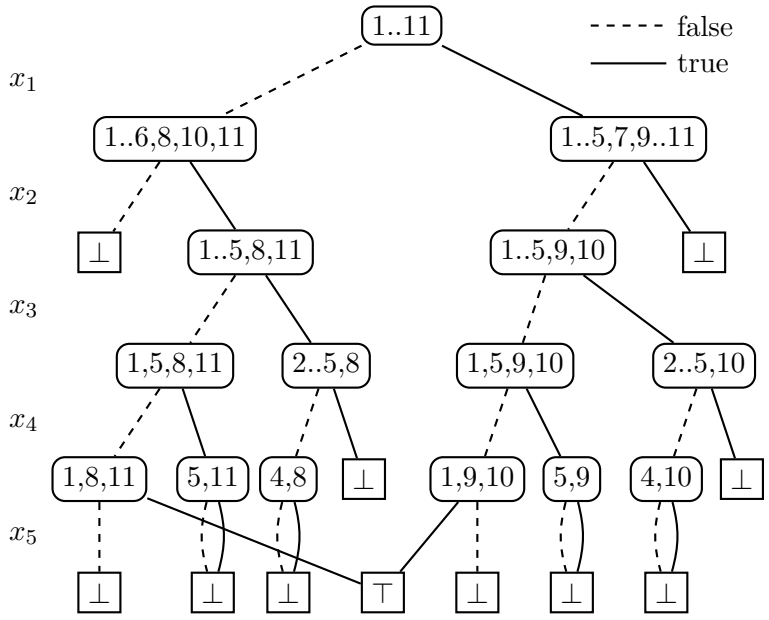

Figure 4.1: The exact BDD for Example 4.1.1. The false sink is drawn multiple times for clarity.

that have not been satisfied by any path from the root to that node. "True" edges are drawn as solid lines, and "false" edges are drawn as dashed lines. ∎

For the construction of an approximate BDD by merging, we need an appropriate merge operation. Since the state of a node is the set of unsatisfied clauses, the appropriate relaxation merge is the intersection operation, and the appropriate restriction merge is the union operation.

4.2 Deducing clauses from BDDs

Katsirelos et al. [54] found that the solution of SAT instances using conflict-directed clause learning (CDCL) often involves bottlenecks: there exist single clauses that the solver must deduce in order to proceed and upon which all future progress depends, but which themselves require significant work to discover.

We propose the use of a BDD representation of a SAT instance to generate clauses. By using a different method to deduce additional clauses and providing them to the SAT solver along with the SAT instance, it is hoped that some of these bottleneck clauses (or some of their prerequisite clauses)

may be learned earlier, thereby speeding up the search for a solution. We show later, in Section 4.3, that BDD-generated clauses can be provably stronger than those produced from a single application of CDCL.

4.2.1 Projections onto single variable domains

One simple way to deduce clauses from a BDD is to project the variable assignments along the satisfying paths in a BDD and to look for variables whose values must be fixed. For instance, in Example 4.1.1 above (see Figure 4.1), we can infer from both feasible paths that we can fix $\overline{x}_3$, $\overline{x}_4$, and x_5. However, in practice we must use approximate BDDs, and this approach does not produce much useful information.

4.2.2 Projections onto multiple variable domains

A second possibility is to derive formulas from projections onto multiple variable domains. For instance, in Example 4.1.1 (Figure 4.1), the variables x_1 and x_2 must take opposite values, which means that it is possible to deduce the formula $(x_1 \wedge \overline{x}_2) \vee (\overline{x}_1 \wedge x_2)$.

Such formulas can be deduced as follows. We define the *intersection* of a collection of conjunctions $A_1, \ldots, A_k$ of literals to be the conjunction of all the literals that appear in all of the conjunctions $A_1, \ldots, A_k$. For example,

$$(x_1 \wedge x_2 \wedge \overline{x}_3 \wedge \overline{x}_4) \cap (x_1 \wedge x_3 \wedge \overline{x}_4 \wedge \overline{x}_5) = x_1 \wedge \overline{x}_4.$$

We also define the *difference* of two conjunctions of literals to be the conjunction of all literals that appear in the first but not the second. For example,

$$(x_1 \wedge \overline{x}_2 \wedge x_3) \setminus (x_2 \wedge x_3 \wedge x_4) = x_1 \wedge \overline{x}_2.$$

The empty conjunction, $\bigwedge \emptyset$, will be denoted $\top$. We use the convention that $\top \wedge A = A \wedge \top = A$ and $\top \vee A = A \vee \top = \top$ for every propositional formula A.

To determine the clauses that can be deduced by projection onto multiple variable domain, we associate with every node n in a BDD two attributes:

- a *required conjunction* $R(n)$, which is the maximal conjunction of literals that is satisfied by every partial assignment corresponding to a path from the root to n; and
- a set $\mathscr{O}(n)$ of *options*, which is a nonempty set of conjunctions of literals such that $R(n) \cap O = \emptyset$ for all $O \in \mathscr{O}(n)$ and every partial assignment

corresponding to a path from the root to n implies $R(n) \wedge O$ for some $O \in \mathscr{O}(n)$.

Theorem 4.2.1. *Let $L = \{n_1, \ldots, n_k\}$ be any set of nodes in the BDD such that every path from the root to the true sink passes through at least one node in L. Then we can deduce the clause*

$$\bigvee_{i=1}^{k} \left[R(n_i) \wedge \bigvee \mathscr{O}(n_i) \right].$$

Proof. This follows from the definitions of $R(n)$ and $\mathscr{O}(n)$. □

Corollary 4.2.2. *Let $L = \{n_1, \ldots, n_k\}$ be any set of nodes in the BDD such that every path from the root to the true sink passes through at least one node in L. Then we can deduce every clause of the form*

$$\bigvee_{i=1}^{k} A_i,$$

where $A_i \in \left\{ R(n_i), \bigvee \mathscr{O}(n_i) \right\}$ for $i = 1, \ldots, k$.

Note that, in particular, we can apply the above results when L is a layer of the BDD. Also observe that the clauses deduced in the corollary are disjunctions of conjunctions of literals. (The clause deduced in the theorem has a more complicated form.)

For an edge e in the BDD pointing from node n_1 to node n_2, let $l(e)$ be the literal corresponding to the variable assignment represented by e (i.e., the edge label of e). For example, the two edges e_1 and e_2 pointing from the root of the BDD to the nodes in the second layer have $l(e_1) = \overline{x}_1$ and $l(e_2) = x_1$, assuming that the BDD uses the standard (lexicographic) variable ordering.

To compute $R(n)$ and $\mathscr{O}(n)$ for the nodes in a BDD, we use the following recursive algorithm.

- The root r has $R(r) = \top$ and $\mathscr{O}(r) = \{\top\}$.
- For an edge e pointing from node n_1 to node n_2, define $R(e) = R(n_1) \wedge l(e)$.
- A non-root node n with incoming edges $e_1, \ldots, e_k$ has

$$R(n) = \bigcap_{i=1}^{k} R(e_i),$$
$$\mathscr{O}(n) = \{\, R(e_i) \setminus R(n) : i = 1, \ldots, k \,\}.$$

However, in practice the clauses generated in this way do not seem to be particularly useful. The quality of these clauses depends very strongly on the variable ordering used in the BDD, and it is not clear how to find a variable ordering that produces good clauses.

4.2.3 Witness clauses from infeasible BDD nodes

A more fruitful approach is to identify the nodes of the BDD from which no path leads to the true sink and, for each such node, to generate a clause that witnesses its infeasibility. As we shall see, we can deduce these clauses systematically from the state information for these nodes.

Definition 4.2.3. A node v in a BDD is *feasible* if there exists a path from v to the true sink. If the BDD is a restricted BDD constructed with the deletion algorithm described in Section 2.3.2, and at least one child node of v was deleted during this algorithm, then we also say that v is feasible, even if there exists no path from v to the true sink in the restricted BDD. If v is not feasible, then it is *infeasible*. An infeasible node v is *maximally infeasible* if v is the root or v has an incoming edge from a feasible node. (Note that a maximally infeasible node may also have incoming edges from infeasible nodes.)

For example, the BDD from Figure 4.1 is redrawn in Figure 4.2 with the infeasible nodes shaded gray. The maximally infeasible nodes are indicated with a double border.

The method described in this section requires a BDD that was constructed using a procedure having a certain property, as described in the following definition.

Definition 4.2.4. Let B be a BDD for a SAT instance that is (a) exact, constructed with the top-down algorithm described in Section 2.1; (b) relaxed, constructed with the merging algorithm described in Section 2.3.1; or (c) restricted, constructed with the deletion algorithm described in Section 2.3.2. Then we say that B is a *shrinking-state BDD*, because neither the top-down construction nor the approximation operation can increase the state of a node: the state of every node is a subset of the state of its parent.

Note in particular that restriction BDDs constructed with the merging algorithm described in Section 2.3.1, using the union operation as the restriction operation, are not shrinking-state BDDs.

For the remainder of this section, we shall assume that we are working with a shrinking-state BDD.

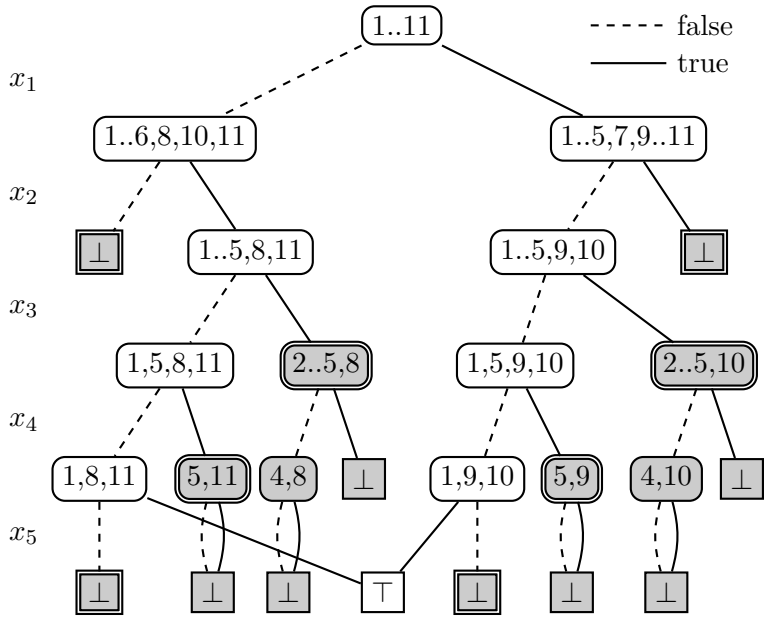

Figure 4.2: The exact BDD for Example 4.1.1, with the infeasible nodes shaded gray and the maximally infeasible nodes indicated with a double border.

We shall deduce clauses from the states of infeasible nodes by applying a sequence of *resolution* steps. Resolution is a commonly used inference rule applied to propositional formulas in conjunctive normal form. The resolution rule, applied to two clauses $x_i \vee P$ and $\overline{x}_i \vee Q$, where P and Q denote disjunctions of literals, is

$$\frac{x_i \vee P \qquad \overline{x}_i \vee Q}{P \vee Q}.$$

Application of this rule is called resolving $x_i \vee P$ and $\overline{x}_i \vee Q$ on the variable x_i, and the resulting clause $P \vee Q$ is called the *resolvent*.

Recall that during the top-down construction of a BDD for a SAT instance, infeasibility of a child node v in layer L_i is detected when the state of v would contain a clause C such that $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$. In other words, the clause C has not yet been satisfied by assignments to the variables $x_1, \ldots, x_{i-1}$, but it contains no variable corresponding to a lower layer of the BDD. When this occurs, we choose one such clause as a witness of the infeasibility of v; we will call this clause $\text{witness}(v)$. According to the BDD construction algorithms discussed previously, the edge from the parent of v should be directed to point to the false sink. However, in an important modification, we will instead construct the node v and treat it as a *leaf node* distinct from other leaf nodes, because it now has additional associated information (its witness clause).

After the BDD construction is complete, we perform a single bottom-up pass to identify all infeasible branch nodes and generate a witness clause for each from the witness clauses of its children. For an infeasible branch node v in layer L_i, we generate the witness clause $\text{witness}(v)$ as follows:

- If v has a child node w such that $\text{witness}(w)$ does not contain the variable x_i, then take $\text{witness}(v) = \text{witness}(w)$.
- Otherwise, the witness clause of the "false" child w_{F} contains the literal x_i and the witness clause of the "true" child w_{T} contains the literal $\overline{x}_i$ (see the proof of Lemma 4.3.4), so resolve $\text{witness}(w_{\text{F}})$ and $\text{witness}(w_{\text{T}})$ on the variable x_i and take the resolvent as $\text{witness}(v)$.

At the end, we output the witness clauses for all maximally infeasible nodes in the BDD.

Example 4.2.5. Continuing the graph-coloring example from earlier (Example 4.1.1), consider the infeasible subtree rooted at the node with state $\{2, 3, 4, 5, 8\}$ in layer L_4 in Figure 4.1. This subtree is redrawn in Figure 4.3.

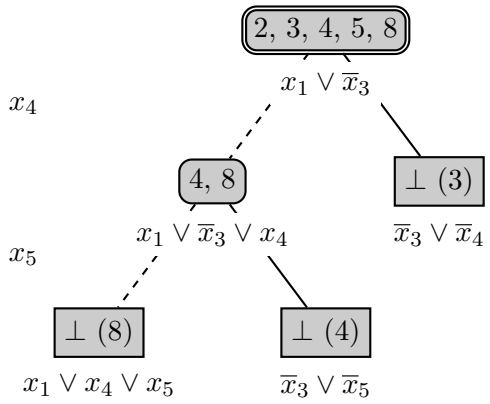

Figure 4.3: Witness clauses generated from the infeasible subtree rooted at the node with state $\{2, 3, 4, 5, 8\}$ in layer L_4.

Setting $x_4 = 0$ satisfies clauses 2, 3, and 5, so the "false" child (i.e., the child along the "false" edge) has state $\{4, 8\}$. However, from this node, setting $x_5 = 0$ means that clause 8 cannot be satisfied, and setting $x_5 = 1$ means that clause 4 cannot be satisfied. Therefore, neither of the children of the node with state $\{4, 8\}$ is feasible, and we have a witness for the infeasibility of each: clause 8 ($x_1 \vee x_4 \vee x_5$) for the "false" child, and clause 4 ($\overline{x}_3 \vee \overline{x}_5$) for the "true" child.

Likewise, returning to the node with state $\{2, 3, 4, 5, 8\}$, if we set $x_4 = 1$ then clause 3 cannot be satisfied, so clause 3 ($\overline{x}_3 \vee \overline{x}_4$) is a witness of the infeasibility of this child.

Now, in our bottom-up pass, we first determine that the node with state $\{4, 8\}$ is infeasible. Both of its child nodes have witness clauses that contain the variable x_5, so we apply the resolution rule to these two witness clauses with respect to x_5 to obtain the clause $x_1 \vee \overline{x}_3 \vee x_4$, which is a witness of the infeasibility of the node with state $\{4, 8\}$. Likewise, the node with state $\{2, 3, 4, 5, 8\}$ is infeasible, so we apply the resolution rule to these two witness clauses with respect to x_4 to obtain the clause $x_1 \vee \overline{x}_3$.

Since the node with state $\{2, 3, 4, 5, 8\}$ is a maximally infeasible node, we produce the clause $x_1 \vee \overline{x}_3$ as output. Note that this is a valid clause for the original SAT instance (it is logically entailed by the instance) because it was produced by the operation of resolution from clauses in the instance and resolution is a sound proof system.

In a similar way, we generate the witness clause $x_2 \vee \overline{x}_3$ for the maximally infeasible node with state $\{2, 3, 4, 5, 10\}$ in layer L_4 in the BDD in Figure 4.1.

This is also a valid clause for the original SAT instance.

Thus we have generated two clauses, $x_1 \vee \overline{x}_3$ and $x_2 \vee \overline{x}_3$, that are not present in the original instance but that are valid: every assignment of values to variables that satisfies the clauses in the original instance must also satisfy these two generated witness clauses. ∎

4.3 Characterization of witness clauses

In this section we examine the strength of the clauses generated by the method described in Section 4.2.3. As in that section, we shall focus only on shrinking-state BDDs.

Definition 4.3.1. Let $y = (y_1, \ldots, y_i)$ be a partial assignment for a SAT instance with variables $x_1, \ldots, x_n$, and let C be a clause on these variables. If some literal of C is satisfied by y (i.e., if C contains a literal x_j with $j \leq i$ such that $y_j = 1$, or C contains a literal $\overline{x}_j$ with $j \leq i$ such that $y_j = 0$), then we say that C is *satisfied* by y; otherwise we say that C is *unsatisfied* by y. If every variable in C is assigned a value by y, but no literal of C is satisfied by these assignments, then we say that C is *falsified* by y.

Note that a clause C can be unsatisfied by a partial assignment y but not falsified by y, if no literal of C is satisfied by y but there remain variables in C that are not assigned values by y.

Lemma 4.3.2. *Let B be a shrinking-state BDD for a SAT instance, let v be a node in B, and let C be a clause in* state(v)*. Then C is unsatisfied by every partial assignment corresponding to a path from the root of B to v.*

Proof. This is true for an exact BDD by the definition of the state function, and it is true for a shrinking-state approximate BDD because the approximation operation cannot introduce new clauses into the state of a node. □

Definition 4.3.3 (Beame et al. [8])**.** A *resolution derivation* of a clause C from a CNF formula F is a sequence $\pi = (C_1, C_2, \ldots, C_s \equiv C)$ of clauses in which each clause C_r either is a clause in F (an *initial* clause) or is derived by applying the resolution operation to clauses C_p and C_q with $p, q < r$ (a *derived* clause).

Lemma 4.3.4. *Let B be a shrinking-state BDD for a SAT instance, let v be an infeasible node in B, and let $C =$* witness(v)*. Then there is a resolution derivation of C from* state(v)*.*

Proof. Let L_i be the layer of B containing v.

In the case when v is a leaf node, C itself is a clause in state(v), so the resolution derivation is simply $\pi = (C)$.

In the case when one of the children w of v has a witness clause C' that does not contain the variable x_i, we have $C = C'$. By induction, there is a resolution derivation π of C' from state(w). Since B is a shrinking-state BDD, we have state(w) $\subseteq$ state(v), and so π is also a resolution derivation of C from state(v).

In the last case, the witness clauses of both children of v contain the variable x_i. Let w_F and w_T be the "false" and "true" children of v, respectively, having witness clauses C_F and C_T. By Lemma 4.3.2, every clause in state(w_F) is unsatisfied by every partial assignment corresponding to a path from the root of B to w_F; in particular, every clause in state(w_F) is unsatisfied by the assignment $x_i = 0$, and therefore no clause in state(w_F) contains the literal $\overline{x}_i$. By induction, there is a resolution derivation π_F of C_F from state(w_F), and the resolvent of any two clauses C' and C'' contains only literals that already appear in either C' or C'', so C_F does not contain the literal $\overline{x}_i$. Likewise, C_T does not contain the literal x_i. Therefore, since both C_F and C_T contain the variable x_i, C_F must contain the literal x_i, and C_T must contain the literal $\overline{x}_i$. Thus it is valid to resolve C_F and C_T on the variable x_i to obtain C, and this, together with the resolution derivations of C_F and C_T from state(w_F) $\subseteq$ state(v) and state(w_T) $\subseteq$ state(v), respectively, gives a resolution derivation of C from state(v). □

Definition 4.3.5. We say that a clause C is *valid* for a propositional formula F if F logically entails C, that is, if $F \wedge C$ has the same set of solutions as F itself.

Corollary 4.3.6. *The witness clause C generated for any infeasible node v in a shrinking-state BDD for a CNF formula F is valid for F.*

Proof. By Lemma 4.3.4, there is a resolution derivation π of C from state(v). As state(v) is a subset of the clauses of F, π is also a resolution derivation of C from F. Therefore, C is valid for F, because resolution is a sound proof system. □

This corollary shows that the witness-clause method generates valid clauses even from approximate BDDs, which is important from a practical standpoint for instances for which an exact BDD would be too large.

The following lemma, theorem, and corollary apply to shrinking-state relaxation BDDs (including exact BDDs).

Lemma 4.3.7. *Let B be a shrinking-state relaxation BDD for a SAT instance. Let v be an infeasible node in layer L_i, and let $C = \text{witness}(v)$. Then $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$.*

Proof. In the case when v is a leaf node, C is a clause in $\text{state}(v)$ that contains no variable corresponding to a lower layer of the BDD, which is to say that $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$.

In the case when one of the children of v (in layer L_{i+1}) has a witness clause C' that does not contain the variable x_i, we have $C = C'$. By induction, $\text{vars}(C') \subseteq \{x_1, \ldots, x_i\}$, but C' does not contain the variable x_i, so $\text{vars}(C) = \text{vars}(C') \subseteq \{x_1, \ldots, x_{i-1}\}$.

In the last case, the witness clause C_F of the "false" child of v contains the literal x_i, and the witness clause C_T of the "true" child of v contains the literal $\overline{x}_i$. By induction, $\text{vars}(C_\text{F})$ and $\text{vars}(C_\text{T})$ are both subsets of $\{x_1, \ldots, x_i\}$. The witness clause C for v is obtained by resolving C_F and C_T on x_i, so $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$. □

Theorem 4.3.8. *Let B be a shrinking-state relaxation BDD for a SAT instance. Let v be an infeasible node in layer L_i, and let $C = \text{witness}(v)$. Then C is falsified by every partial assignment corresponding to a path from the root of B to v.*

Proof. By Lemma 4.3.7, $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$. Therefore, every partial assignment corresponding to a path from the root to v assigns a value to every variable in C, so such a partial assignment must either satisfy C or falsify it.

Suppose for the sake of contradiction that some such partial assignment y satisfies C. Then y satisfies some literal l of C. By Lemma 4.3.4, there is a resolution derivation of C from $\text{state}(v)$, so there exists a clause C' in $\text{state}(v)$ that contains the literal l. But then y satisfies C', which contradicts Lemma 4.3.2. □

In particular, for an infeasible node v in a relaxation BDD, $\text{witness}(v)$ is never a tautology, nor does it contain any variable that is assigned the value 0 in some path from the root to v and the value 1 in some other such path.

Corollary 4.3.9. *Let B be a shrinking-state relaxation BDD for a SAT instance. Let v be an infeasible node in B, and let $C = \text{witness}(v)$. Then C witnesses the infeasibility of v.*

Proof. If a partial assignment y corresponding to a path from the root to v is extended to a full assignment (y, z), then (y, z) also falsifies C. □

In other words, Corollary 4.3.9 says that any (complete) assignment of values to variables corresponding to a path that passes through v falsifies witness(v).

The next theorem shows that the set of clauses produced by the method in Section 4.2.3 for an exact BDD is a reformulation of the original instance.

Theorem 4.3.10. *Let F be a CNF formula and let B be a shrinking-state exact BDD for F. Let U be the set of maximally infeasible nodes in B, and let*

$$G = \bigwedge_{v \in U} \text{witness}(v).$$

Then G is a reformulation of F, i.e., G has the same set of solutions as F.

Proof. By Corollary 4.3.6, all witness clauses of nodes in U are valid for F. Therefore, if y is a solution of F, it must satisfy all of these witness clauses, so y is also a solution of G. On the other hand, if y is not a solution of F, then the corresponding path P in B must end at an infeasible leaf node, and so there must be a maximally infeasible node v along P. By Corollary 4.3.9, y falsifies witness(v), so y is also not a solution of G. □

In the remainder of this section, we explore how BDD-guided clause generation relates to propagation and inference techniques used in today's SAT solvers.

Definition 4.3.11. *Unit propagation* on a CNF formula is the process of identifying, if there is one, a clause that contains only one literal l, setting l to true, simplifying the formula by deleting $\bar{l}$ from all clauses and removing all clauses containing l, and repeating. Unit propagation is said to result in a *conflict* if it generates the empty clause Λ.

For a clause C and a set S of variables, we use the notation $C - S$ to denote the clause obtained by deleting from C all literals involving variables in S. Observe that if v is a node in layer L_i of a BDD, then any clause C in state(v) is unsatisfied by every partial assignment y corresponding to a path from the root to v (by Lemma 4.3.2), which is an assignment of values to the variables $x_1, \ldots, x_{i-1}$, so C will be satisfied by a full assignment (y, z) if and only if z satisfies $C - \{x_1, \ldots, x_{i-1}\}$. Consequently, if unit propagation on $C - \{x_1, \ldots, x_{i-1}\}$ results in a conflict, then no such partial assignment y has a feasible completion z.

Definition 4.3.12. Given a BDD B, a *unit-propagated BDD*, denoted B_{up}, is the BDD obtained from B by replacing with a leaf node all nodes v such

that unit propagation on $\{ C - \{x_1, \ldots, x_{\text{layer}(v)-1}\} : C \in \text{state}(v) \}$ results in a conflict.

To extend the witness-clause method to unit-propagated BDDs, we need a way to generate witness clauses for nodes deduced to be infeasible by unit propagation. This can be done as shown in Algorithm 5.

Algorithm 5 Unit propagation on node v with witness clause generation

```
1:  i := layer(v)
2:  J := {1, 2, ..., |state(v)|}                    ▷ indices of unsatisfied clauses
3:  for all C_j ∈ state(v) do
4:      R[j] := C_j − {x_1, ..., x_{i−1}}            ▷ working version of clause
5:      W[j] := C_j                                   ▷ tentative witness clause
6:  end for
7:  while ∃ j ∈ J such that |R[j]| = 1 do
8:      l := unique literal in R[j]
9:      J := J \ {j}                                  ▷ set l to true, remove unit clause
10:     for all k ∈ J do
11:         if l ∈ R[k] then
12:             J := J \ {k}                          ▷ remove satisfied clause
13:         else if l̄ ∈ R[k] then
14:             R[k] := R[k] \ l̄                      ▷ remove l̄ from clause
15:             W[k] := resolvent of W[j] and W[k] on var(l)
16:             if R[k] = Λ then                      ▷ unit propagation infers a conflict
17:                 report infeasibility of v with witness clause W[k]
18:             end if
19:         end if
20:     end for
21: end while
```

The loop on lines 7–21 of Algorithm 5 maintains the following two invariants:

- For all $j \in J$, $R[j] = W[j] - \{x_1, \ldots, x_{i-1}\}$. This justifies the use of the resolution operation on line 15: the clause $R[j]$ contains the literal l, and the clause $R[k]$ contains the literal $\bar{l}$, so it is valid to resolve $W[j]$ and $W[k]$ on var(l).
- For all $j \in J$, there is a resolution derivation of $W[j]$ from state(v), by construction. Therefore, the witness clause reported in line 17 satisfies Lemma 4.3.4 (for a shrinking-state BDD).

Note also that if a witness clause $W[k]$ is reported in line 17, then $R[k] = \Lambda$, so the first invariant above implies that $\text{vars}(W) \subseteq \{x_1, \ldots, x_{i-1}\}$, and therefore the witness clause $W[k]$ satisfies Lemma 4.3.7 (for a shrinking-state relaxation BDD).

Consequently, all of the preceding theorems in this section still hold for unit-propagated BDDs with witness clauses generated using Algorithm 5.

Definition 4.3.13 (Atserias et al. [5], Pipatsrisawat and Darwiche [81])**.** A nonempty clause C is said to be *absorbed* by a CNF formula F if for every literal l in C, performing unit propagation on F starting with all literals of C except l set to false either infers l or infers a conflict.

The meaning of this definition is that a clause C is absorbed by F if F and $F \wedge C$ have identical entailment power with respect to unit propagation, that is, any clause that can be derived from $F \wedge C$ using unit propagation can also be derived from F alone.

Pipatsrisawat and Darwiche [81] showed that the conflict-directed clause learning (CDCL) mechanism in SAT solvers always produces clauses that are not absorbed by the current theory, that is, by the set of initial clauses of F and those learned thus far during the search. As we show next, this property also holds for witness clauses generated at branch nodes by the BDD method applied to F.

Theorem 4.3.14. *Let F be a CNF formula, and let B_{up} be a unit-propagated shrinking-state relaxation BDD for F. Let v be a maximally infeasible branch node in layer L_i of B_{up}, and let $C = \text{witness}(v)$. If C is not the empty clause Λ, then C is not absorbed by F.*

Proof. We show that there exists a literal l in C such that performing unit propagation on F starting with all literals of C except l set to false infers neither l nor a conflict.

Since v is a maximally infeasible node, either v is the root or v has an incoming edge from a feasible node u. But if v is the root, then as a consequence of Lemma 4.3.7 we must have $C = \Lambda$. So assume that v has an incoming edge from a feasible node u.

By Lemma 4.3.7, $\text{vars}(C) \subseteq \{x_1, \ldots, x_{i-1}\}$, and by Theorem 4.3.8, C is falsified by every partial assignment corresponding to a path from the root of B_{up} to v. If C does not contain the variable x_{i-1}, then C must therefore be falsified by every partial assignment corresponding to a path from the root to u. But C is valid for F by Corollary 4.3.6, so this would imply that u is infeasible, which is a contradiction. Hence, C must contain the

variable x_{i-1}. Let l be the literal in C that is either x_{i-1} or $\overline{x}_{i-1}$. (By Theorem 4.3.8, $l = x_{i-1}$ if v is the "false" child of u, and $l = \overline{x}_{i-1}$ if v is the "true" child of u.)

Let y be a partial assignment corresponding to a path P from the root to v through u, and let z be the partial assignment corresponding to the portion of P from the root to u. If unit propagation on F starting from z infers a conflict, then u would be a leaf node in B_{up}, which it is not. If unit propagation on F starting from z infers l, then unit propagation on F starting from y infers a conflict, so v would be a leaf node in B_{up}, which it is not. Therefore unit propagation on F starting from z (i.e., setting all literals of C except l set to false) infers neither l nor a conflict, so C is not absorbed by F. □

This theorem establishes that our clause generation approach effectively produces clauses that provide useful information not already captured by unit propagation inference on F. Note that if $C = \Lambda$ then we have deduced infeasibility.

While the solver can eventually learn any clause entailed by F (including the empty clause Λ if F is unsatisfiable) by using a series of potentially exponentially many applications of the CDCL mechanism, we show below that any clause that it can learn with *one* application of conflict analysis starting from a CNF formula F can also be learned as a BDD-generated witness clause from F.

In order to more precisely characterize these witness clauses, we first define several restrictions on the structure of resolution proofs.

Definition 4.3.15 (see Beame et al. [8])**.** A resolution derivation $\pi = (C_1, C_2, \ldots, C_s \equiv C)$ of a clause C from a CNF formula F is called

- *tree-like* if every nonempty derived clause C_r is used exactly once;
- *regular* if every variable is resolved upon at most once along any "path" in the proof from an initial clause to C, allowing (restricted) reuse of derived clauses;
- *linear* if every derived clause C_r is obtained by resolving C_{r-1} with C_q for some $q < r - 1$;
- *ordered* (or a *Davis–Putnam* resolution derivation) if it is regular and every sequence of resolved variables along any path from an initial clause to C respects the same ordering on the variables;

- *trivial* if all variables resolved upon are distinct and each C_r, for $r \geq 3$, is either an initial clause or is derived by resolving C_{r-1} with an initial clause.

Note that a trivial derivation is tree-like, regular, linear, and ordered.

Each of these restricted resolution schemes is sound and complete as a proof system, but they differ in their efficiency (measured by the number of clauses required in a resolution derivation).

Proposition 4.3.16 (Bonet and Galesi [18], Bonet et al. [19], Buresh-Oppenheim and Pitassi [23])**.** *Regular, linear, and ordered resolution are each exponentially stronger than tree-like resolution.*

Proposition 4.3.17 (Alekhnovich et al. [3], Bonet et al. [19])**.** *Tree-like, regular, and ordered resolution are each exponentially weaker than unrestricted resolution.*

Proposition 4.3.18 (Beame et al. [8, Proposition 4])**.** *Any conflict clause produced by CDCL using any clause learning scheme can be derived from initial and previously derived clauses using a trivial resolution derivation.*

This means that each intermediate clause C_{r+1} in a CDCL derivation τ is obtained by resolving C_r with a clause of F and that the sequence of variables resolved upon in τ consists of all distinct variables, thereby giving τ a "ladder-like" structure.

Theorem 4.3.19. *For any clause C learned from one application of SAT conflict analysis on F using any clause learning scheme, there exists a variable ordering under which a top-down approximate BDD for F of width at most $2^{|C|}$ generates a clause $C' \subseteq C$.*

Proof. Let τ be the CDCL derivation of C from F, and let σ be the sequence of variables resolved upon in τ. The top-down (partial) variable ordering we use is as follows: first the variables that appear in C (in any order), followed by the variables in the reverse order of σ. The first $|C|$ variables result in a BDD B of width at most $2^{|C|}$. Let v be the node of B in layer $L_{|C|+1}$ at which all literals of C are falsified. When expanding B from v, the ladder-like structure of τ guarantees that at least one branch on the variables in σ can be labeled directly by a clause of F that is falsified. The corresponding lower part of B starting at v is thus of width 1. For the remaining $2^{|C|} - 1$ nodes of B in layer $L_{|C|+1}$, we construct an approximate lower portion of the BDD such that the overall width does not increase. This makes the overall width of B at most $2^{|C|}$.

While B may have several infeasible nodes, the node v in layer $L_{|C|+1}$ is guaranteed by the derivation τ to be infeasible. Let y be a partial assignment corresponding to a path P from the root of B to v. Let u be a maximally infeasible node on P, and let $C' = \text{witness}(u)$. By Theorem 4.3.8, C is falsified by y, and C' is falsified by the partial assignment z corresponding to the portion of P from the root to u. By construction, C contains all $|C|$ literals in y, and by Lemma 4.3.7, C' contains a subset of the literals in z and hence in y. It follows that $C' \subseteq C$. □

The above reasoning can be extended to construct an *exact* BDD that generates a subclause of C. However, the width of such a BDD will depend not only on $|C|$ but also on the number of resolution steps involved in conflict analysis during the derivation of C.

Theorem 4.3.20. *Witness clauses generated from F correspond to regular and ordered resolution derivations starting from the clauses of F.*

Proof. It is easily seen that the resolution operations performed during clause generation from a BDD respect, by construction, the restrictions of being regular and ordered. Hence, any witness clause C can be derived using regular and ordered resolution starting from F.

On the other hand, let τ be any regular and ordered resolution derivation of C starting from F. An argument similar to the one in the proof of Theorem 4.3.19 can be used to show that there exists a natural variable ordering (namely, first branch on the variables of C, then follow the top-down variable ordering imposed by τ) under which the top-down BDD B for F contains a node v such that any path from the root of B to v falsifies all literals of C. As before, witness clauses for B may not directly include C as is, but the witness clause C' associated with a maximally infeasible node u with v as a descendant would be a subclause of C. □

We recall again the resolution-based characterization of CDCL clauses, namely, those that can be derived using trivial resolution (i.e., tree-like, regular, linear, and ordered resolution). This results in linear-size resolution derivations and thus forms a strict subset of all possible derivations that are regular and ordered, because not all regular and ordered derivations are tree-like and linear. Theorem 4.3.20 therefore implies the following result.

Corollary 4.3.21. *There exist BDD-generated witness clauses that cannot be derived using one application of SAT conflict analysis.*

4.4 Implementation and experimental results

We implemented the clause generation algorithm described in Section 4.2.3 in C++, as a program called Clausegen. See Chapter 5 for a discussion of several implementation details.

To demonstrate our method, we considered SAT instances produced from randomly generated bipartite graph matching problems, with 15 vertices on each side, in which a random subset of 10 vertices on one side is adjacent to only 9 vertices on the other side, so that the graph fails to satisfy Hall's condition, thereby making the SAT instance unsatisfiable. We preprocessed the instance with SatELite 1.0 [35] (using the `+pre` option) and used MiniSat 2.2.0 [36] as the SAT solver (with `-rnd-freq=0.01`). Because MiniSat uses a nondeterministic algorithm, it was run 20 times for each test with different random seeds, and the results were averaged. The experiments were run on an Intel Xeon E5345 at 2.33 GHz with 24 GiB of RAM, running Ubuntu 12.04.5.

For a representative instance of this type, with 225 variables and 748 clauses (80 variables and 405 clauses after preprocessing), MiniSat made 864,930 decisions and encountered 714,625 conflicts on average. (This instance, hall-set-15-10_01.cnf, is given in Appendix A.)

In Figure 4.4 we show the results of appending the clauses produced by Clausegen before the instance is given to MiniSat. As the maximum BDD width is increased from 10 to 10,000, thus yielding more accurate approximate BDDs, the numbers of decisions and conflicts encountered by MiniSat decrease. The clauses generated at BDD width 10,000 produced an improvement in these metrics by over 75% in comparison with the original instance: MiniSat averaged 212,158 decisions and 178,101 conflicts.

However, we do not see a corresponding improvement in the running time of MiniSat. The stacked area plot in Figure 4.5 shows the running time of Clausegen and MiniSat as the BDD width is increased. On the original instance, MiniSat required an average of 7.83 seconds; this time increased to 17.65 seconds when the clauses generated at BDD width 10,000 were added. The number of generated clauses increases linearly with the BDD width, from 12 clauses at width 10 to 9745 clauses at width 10,000, as seen in Figure 4.6. The clauses generated at width 10,000 have an average length of 11.8, compared to an average length of 2.1 in the original instance.

Figures 4.7 and 4.8 show our results for another instance, countingclqcolor-unsat-set-b-clqcolor-08-06-07.sat05-1257.reshuffled-07.cnf, from the SAT Challenge 2012 Hard Combinatorial SAT+UNSAT benchmark instan-

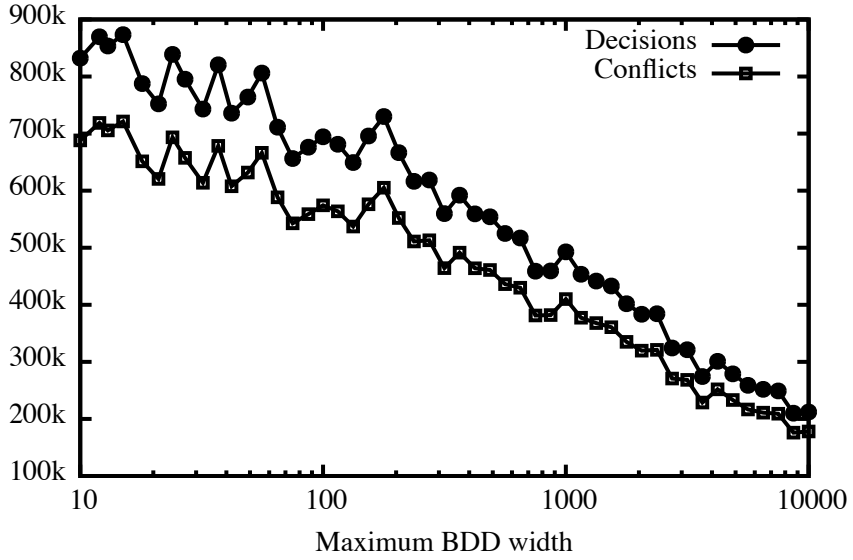

Figure 4.4: Numbers of decisions and conflicts encountered by MiniSat on an unsatisfiable bipartite matching instance.

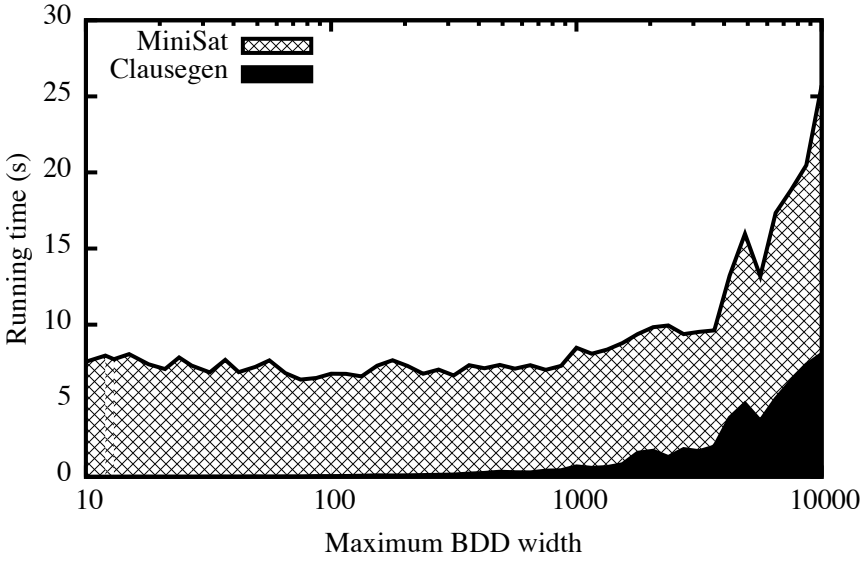

Figure 4.5: Running time of Clausegen and MiniSat on an unsatisfiable bipartite matching instance.

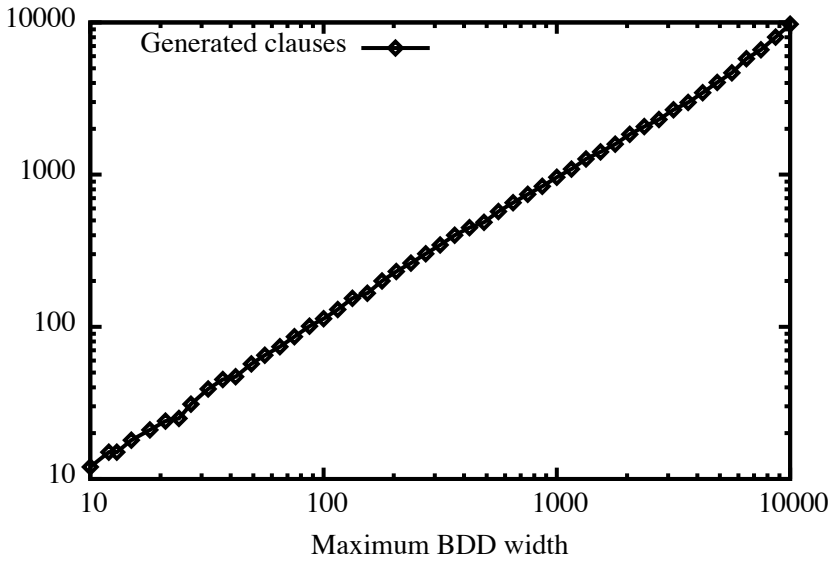

Figure 4.6: Number of witness clauses generated at various BDD widths for an unsatisfiable bipartite matching instance.

ces[1] and due to Ashish Sabharwal. This instance has 132 variables and 1527 clauses of average length 2.9 (117 variables and 1599 clauses of average length 4.3 after preprocessing with SatELite) and is also unsatisfiable; it represents a graph coloring instance with a hidden clique that is larger than the number of colors available. MiniSat averaged 2,072,107 decisions and 1,511,029 conflicts for the original instance, taking 14.18 seconds on average. When the 3255 clauses of average length 8.7 produced by Clausegen at BDD width 10,000 were added, the average numbers of decisions and conflicts decreased to 713,718 and 515,514, respectively, and the average running time of MiniSat decreased to 6.34 seconds. The minimum total running time of Clausegen and MiniSat together was achieved at a BDD width of 464; Clausegen took 0.57 seconds to generate 340 clauses of average length 8.7, and MiniSat averaged 1,351,691 decisions and 972,674 conflicts, taking 9.14 seconds on average to solve the instance, for a total average solving time of 9.71 seconds (an improvement of 31.5% over the original instance).

Our results for hole10.cnf from the SATLIB library [48] are shown in Figures 4.9 and 4.10. This instance, due to John Hooker, represents an unsatisfiable pigeonhole instance in which 11 pigeons are to be placed into

[1] See `http://baldur.iti.kit.edu/SAT-Challenge-2012/downloads.html`.

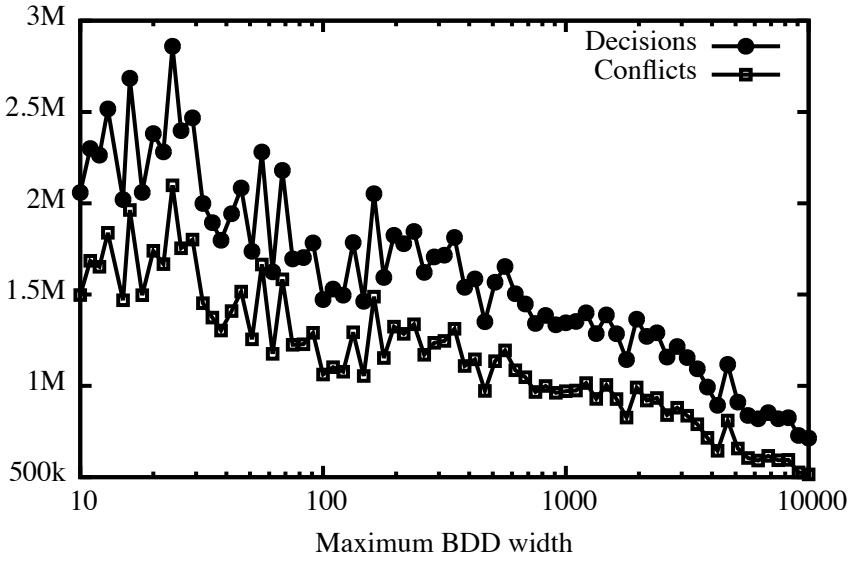

Figure 4.7: Numbers of decisions and conflicts encountered by MiniSat on counting-clqcolor-unsat-[...].cnf from SAT Challenge 2012.

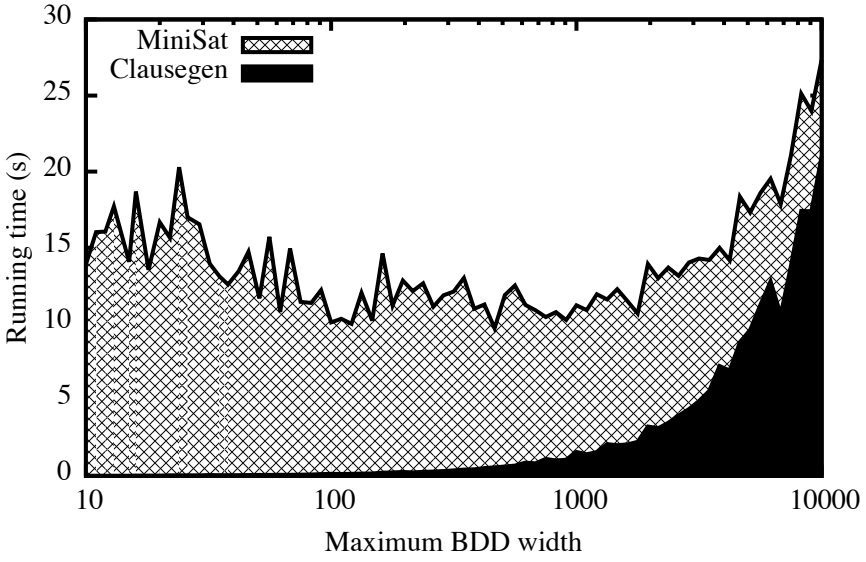

Figure 4.8: Running time of Clausegen and MiniSat on counting-clqcolor-unsat-[...].cnf from SAT Challenge 2012.

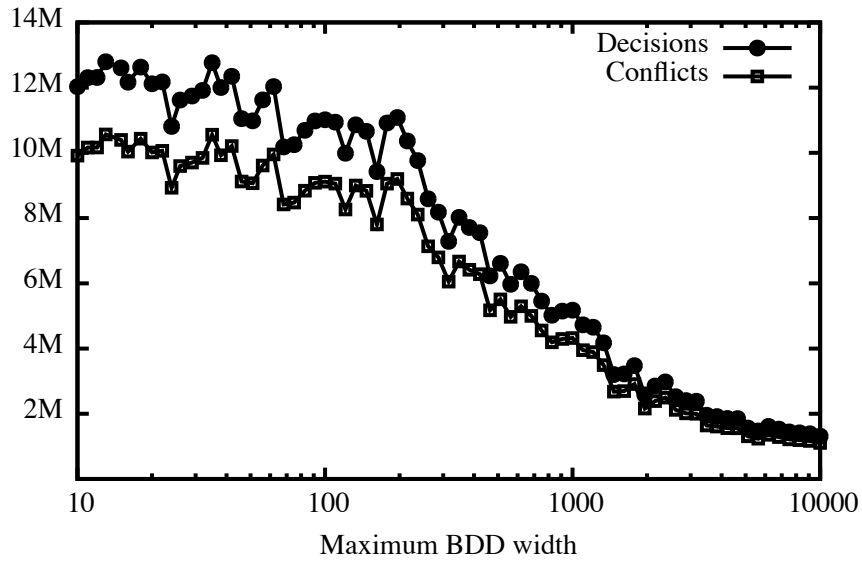

Figure 4.9: Numbers of decisions and conflicts encountered by MiniSat on hole10.cnf from SATLIB.

10 holes without placing two pigeons in the same hole. The original instance has 110 variables and 561 clauses of average length 2.2 (550 clauses of length 2 and 11 clauses of length 10). MiniSat averaged 12,416,393 decisions and 10,230,929 conflicts on the original instance, taking 177.78 seconds. Clausegen produced 15,440 clauses in 5.88 seconds at BDD width 10,000; when these clauses are added to the original instance, the numbers of decisions and conflicts decrease to 1,311,429 and 1,102,655, respectively (improvements of 89.4% and 89.2%). The minimum total running time of Clausegen and MiniSat together was achieved at BDD width 1957. At this width, Clausegen took 1.05 seconds to generate 3512 clauses, and MiniSat took an average of 110.31 seconds to solve the augmented instance, encountering 2,580,030 decisions and 2,158,505 conflicts; the total solving time was 111.35 seconds, an improvement of 37.4% over the original instance.

Figures 4.11 and 4.12 show the results of applying Clausegen to a SAT instance generated from a certain biconditional formula. Groote and Zantema [44] gave a construction for a family of unsatisfiable biconditional formulas that require exponentially long resolution proofs of unsatisfiability. The formula $\neg[S_4]$ produced on the fourth iteration of this construction was converted to a CNF formula by the Tseitin transformation described

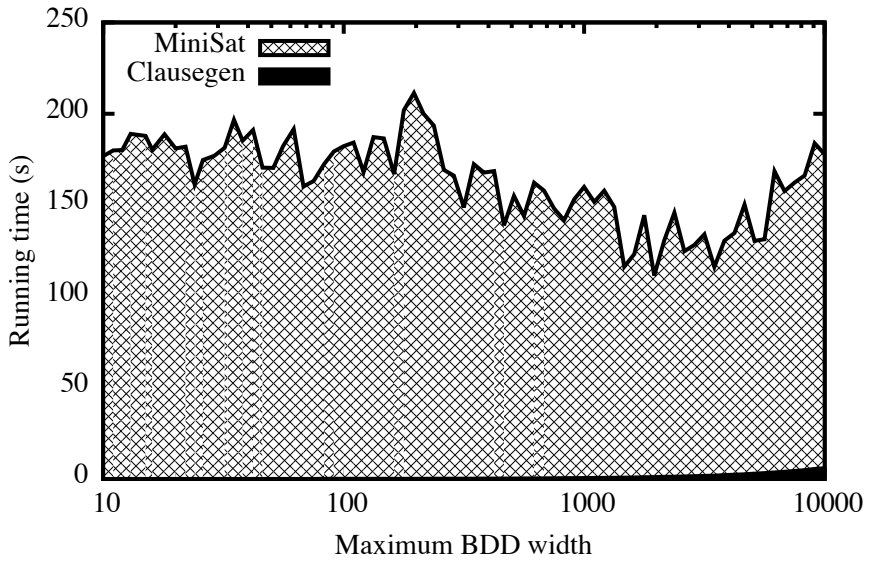

Figure 4.10: Running time of Clausegen and MiniSat on hole10.cnf from SATLIB.

in Section 6.1.1. The resulting SAT instance (gz4.cnf; see Appendix A) has 95 variables and 253 clauses (252 clauses of length 3 and one clause of length 1) and is unsatisfiable by construction. In our tests, MiniSat took an average of 1.80 seconds to deduce unsatisfiability, after 842,330 decisions and 459,076 conflicts.

The exact BDD for this instance has width 2048. As can be seen in Figure 4.11, as the width of the approximate BDD approaches 2048, the numbers of decisions and conflicts encountered by MiniSat decrease. When the maximum-width parameter of Clausegen is 2048 or larger, an exact BDD is constructed, from which Clausegen can deduce unsatisfiability and return the empty clause. The empty clause allows MiniSat to terminate immediately. Therefore, for a sufficiently large BDD width, the instance is solved completely by Clausegen, taking 0.50 second on average (an improvement of 72.2% over MiniSat).

4.5 Summary

In this chapter we presented a BDD-based algorithm to deduce valid clauses from an instance of the Boolean satisfiability problem. Section 4.1 described

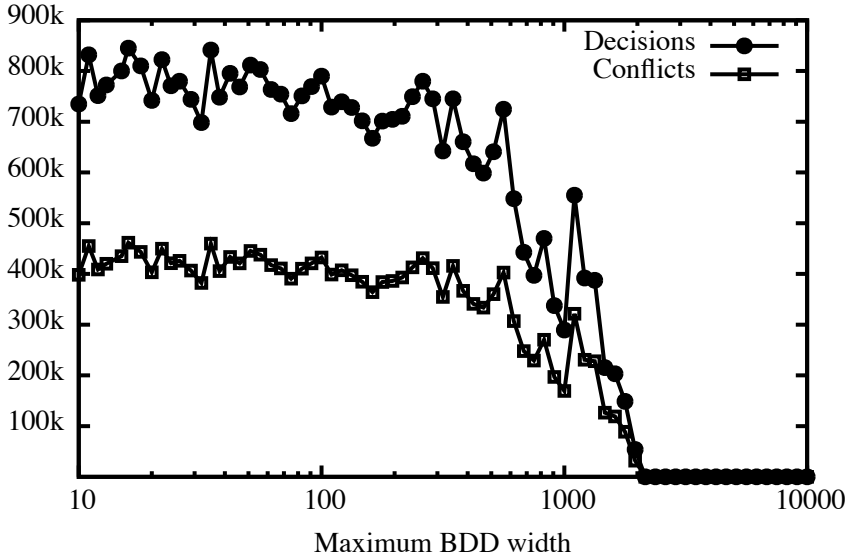

Figure 4.11: Numbers of decisions and conflicts encountered by MiniSat on an unsatisfiable biconditional formula.

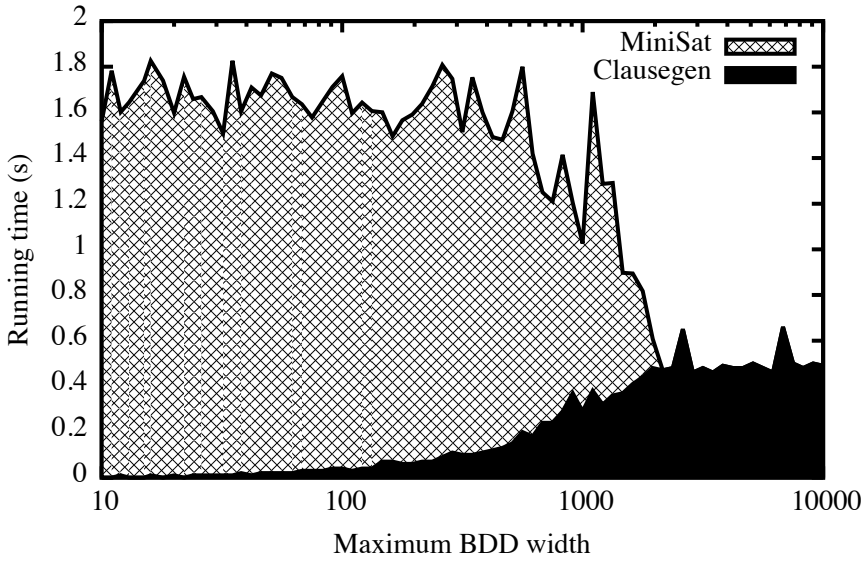

Figure 4.12: Running time of Clausegen and MiniSat on an unsatisfiable biconditional formula.

the BDD representation of SAT instances. In Section 4.2 we presented three techniques for deducing clauses from these BDDs, including an algorithm to generate clauses that witness the infeasibility of BDD nodes. We characterize these witness clauses in Section 4.3; in particular, we showed that for an exact BDD the full set of generated witness clauses is a reformulation of the instance and the witness clauses are still valid when generated from shrinking-state approximate BDDs. When a unit-propagated BDD is used, the generated witness clauses are not absorbed by the clauses of the original instance, any clause that can be learned from one application of conflict analysis can also be generated by our witness-clause algorithm, and our witness-clause algorithm can generate strictly stronger clauses than one application of conflict analysis. The experimental results in Section 4.4 demonstrated that the clauses generated by our method can be effective in reducing the numbers of conflicts and decisions encountered by a SAT solver.

Chapter 5

Implementation considerations

In this chapter we briefly discuss a few of the implementation considerations necessary to put the methods of Chapters 3 and 4 into practice.

5.1 Implementation of bin packing MDDs

5.1.1 Variable ordering

The variable ordering used in a BDD or MDD can have a very significant effect on its size. Behle [10] investigated the optimal variable ordering problem for threshold BDDs. Bollig and Wegener [17] showed that in general the problem of determining whether a given variable ordering of a BDD can be improved is NP-complete.

It is desirable to have a variable ordering that results in a small MDD, even if approximate MDDs are to be used, because the structure of the approximate MDD will be closer to the structure of the exact MDD and will give better results.

For the multidimensional bin packing problem, we take a simple heuristic approach. We observe that identifying dead bins and free bins is beneficial, and we would like to make such identifications as soon as possible. If we pack the largest items first, then the total size of the remaining unpacked items will decrease quickly in the beginning, which suggests that we may reach free bins early; additionally, we will tend to fill bins quickly in the beginning, which suggests that we may exhaust the bins' capacity quickly and reach dead bins early. However, these ideas are somewhat contradictory,

and the latter idea is opposed by the observation that the unpacked items will be small, so they can fit into small spaces.

We therefore use an "interleaved" ordering, in which the largest item is packed first, then the smallest item, then the second largest item, then the second smallest item, and so on, packing the median-sized item last. (Our item sizes $(s_{i,1}, \ldots, s_{i,d})$ are multidimensional, so we use the total size of the item in all dimensions: $\sum_{k=1}^{d} s_{i,k}$.) This ordering seemed to work well for our experimental instances. This straightforward approach means that we can implement variable ordering by sorting the items in this manner as a preprocessing step.

5.1.2 Precomputation

Recall from Section 3.4 that it can be profitable to round down ullages in the states of the ullage MDD representation: Let $u_{j,k}$ denote the ullage of bin j, in the kth dimension, after we have placed items 1 through i into bins. Let a denote the greatest possible sum of a subset of the sizes of items $i+1$ through n, in the kth dimension, that does not exceed $u_{j,k}$. If $a < u_{j,k}$, then we may consider the ullage of bin j, in the kth dimension, to be a rather than $u_{j,k}$ without changing the set of feasible completions.

If the order of the items is fixed, then the relevant sets of possible sums of remaining items can be computed once at the beginning of the MDD construction in $O(nc_{\max}^2)$ time, where $c_{\max}$ is the largest bin capacity in a single dimension.

5.2 Implementation of BDDs for SAT instances

5.2.1 Data structures

The `boost::dynamic_bitset<>` data structure from the Boost library[1] is an implementation of a set of bits whose size can be set at run time. This data structure is very useful for representing various sets of clauses; for example, the state of a node is a set of unsatisfied clauses. The clauses themselves, which are not changed after the instance has been read from input, can be stored in an array or a vector. The dynamic bitset then represents a subset of these clauses. The implementation of this class supports efficient compu-

[1] Documentation for version 1.54.0 of the `boost::dynamic_bitset<>` package, which is the version we used in our experiments, is available online at `http://www.boost.org/doc/libs/1_54_0/libs/dynamic_bitset/dynamic_bitset.html`.

tation of many set operations, such as union, intersection, complement, and nonemptiness testing.

5.2.2 Variable ordering

Clausegen supports several variable ordering heuristics. For our experiments, we used the following heuristic: each variable is assigned a score, computed as the quotient between the number of clauses containing the variable and the average arity of those clauses, and the variables are sorted in decreasing order according to this score, so that higher-scoring variables (that is, variables that appear in many mostly short clauses) correspond to layers nearer the top of the BDD. This heuristic is enabled in Clausegen by specifying `ccaa` for the `-ordering` option (`ccaa` stands for "clause count/ average arity").

Other variable ordering heuristics supported by Clausegen include `lex` (the lexicographic variable ordering: x_1, x_2, x_3, etc.) and `random` (a random variable ordering).

In each of these schemes, the variable ordering is determined before the BDD is constructed (as opposed to an ordering being determined dynamically during the top-down construction). This allows us to do several computations during preprocessing phases, thereby avoiding recomputation, as described in the next section.

5.2.3 Preprocessing

It is useful to precompute the sets of clauses that are satisfied by each possible variable assignment. For an instance with n variables, this requires the computation of $2n$ subsets of clauses (represented by dynamic bitsets). This allows us to easily and efficiently implement the child state function: if v is a node in the BDD under construction, x_i is the variable on which v branches (i.e., the two outgoing edges from v correspond to the assignments $x_i = 0$ and $x_i = 1$), and S_0 is the set of clauses that are satisfied by the assignment $x_i = 0$, then the state of the "false" child of v is $\text{state}(v) \setminus S_0$. Similarly, if S_1 is the set of clauses that are satisfied by the assignment $x_i = 1$, then the state of the "true" child of v is $\text{state}(v) \setminus S_1$.

Additionally, once the variable ordering has been determined, we can precompute, for each layer of the BDD, the set of clauses that can be satisfied after that layer (that is, the set of clauses containing at least one variable that appears later in the variable ordering). These sets allow for efficient testing of infeasibility: if the child state function returns the state X for

a child node w, A is the set of clauses that can be satisfied after the layer containing w, and $X \cap A = \emptyset$, then w is infeasible.

If a unit-propagated BDD is being constructed, it is also useful to precompute subsets of clauses that can trigger unit propagation at each layer of the BDD. This is explained in greater detail in Section 5.2.5.

5.2.4 Merging heuristics

The construction of a relaxation BDD via merging requires a rule for determining which nodes to merge in a layer that exceeds the maximum width. Since unsatisfied clauses lead to infeasibility, and our method generates clauses from infeasible subtrees, the following merging rule was used for our experiments: if a constructed layer exceeds the maximum width W, sort the nodes by the number of unsatisfied clauses in their states, preserve the $W-1$ nodes with the greatest number of unsatisfied clauses, and merge the other nodes into a single node. (The state of the resulting node is the intersection of the states of the nodes that were merged.) Merging rules similar to this one have been applied before in the context of optimization and scheduling, for example by Cire and van Hoeve [28]. This merging rule is enabled in Clausegen by specifying `unsat` for the `-merger` option.

Other merging heuristics supported by Clausegen include `sat` (which preserves the states with the fewest unsatisfied clauses) and `random` (which merges pairs of nodes at random until the width of the layer is no greater than W).

5.2.5 Unit propagation

Given a fixed variable ordering $(x_{i_1}, \ldots, x_{i_n})$ and a clause C, define

$$u(C) = \big\{\, k \in \{1, \ldots, n\} : \big|C - \{x_{i_1}, \ldots, x_{i_{k-1}}\}\big| = 1 \,\big\}.$$

In other words, $u(C)$ is the set of integers k such that removing all variables in the set $\{x_{i_1}, \ldots, x_{i_{k-1}}\}$ from C leaves a unit clause (a clause with exactly one literal). The set $u(C)$ contains the element 1 if and only if C itself is a unit clause.

The intuitive meaning of $u(C)$ is that if C is not yet satisfied at a node in layer L_k of the BDD, and $k \in u(C)$, then C has "one chance left"—there is only one variable in C corresponding to a lower layer of the BDD. Consequently, at a node v in layer L_k, $k \in u(C)$, at which C has not yet been satisfied, we can treat C as a unit clause, because all but one of its literals has been falsified by any partial assignment corresponding to a path

from the root to v, and therefore we can apply unit propagation to state(v), beginning by setting the sole remaining unfalsified literal of C to true.

For each layer L_i of the BDD, we can precompute the subset of clauses C such that $p(C) = i$. These are the clauses that can trigger unit propagation at layer L_i if they have not yet been satisfied. Call this subset T_i; it can be represented with a dynamic bitset, like other subsets of clauses. If v is a node in layer L_k, then we need to run unit propagation (Algorithm 5) on v only if state(v) $\cap\, T_k \neq \emptyset$.

Chapter 6

SAT decomposition

There is a practical limit on the size of the SAT instances to which the techniques in Chapter 4 can be applied. Larger instances require weaker approximate BDDs, and as the approximation becomes weaker, the quality of the deduced clauses decreases. Therefore, it is useful to be able to decompose a SAT instance into smaller subinstances (i.e., subsets of clauses) and work with the subinstances instead of the whole instance. Note that valid clauses deduced from a subset of the clauses of an instance are valid for the instance as a whole.

In this chapter we propose three methods for extracting structured subinstances from a SAT instance. The first, described in Section 6.1.2, is based on recognizing subsets of clauses produced by the Tseitin transformation, which is a widely used procedure to reformulate an arbitrary propositional formula as a formula in conjunctive normal form. The second, described in Section 6.2, is a general technique to determine subinstances by analyzing the connectivity of the constraint graph. The third, described in Section 6.3, produces subinstances by modeling the SAT instance as a resistive electrical network. The results of this chapter are still rather preliminary; more work remains to be done to develop these ideas into practicable algorithms.

6.1 Tseitin clauses

In general, rewriting a propositional formula in conjunctive normal form without the introduction of new variables produces a CNF formula that is exponentially long. In 1968, Tseitin [95] gave a construction to transform any propositional formula into a CNF formula with only a linear increase in length by using auxiliary variables.

6.1.1 The Tseitin transformation

The idea of the Tseitin transformation is to introduce a new variable for each subformula of the original formula and to emit clauses that ensure this variable is true if and only if the corresponding subformula is true. In addition, a unit clause (i.e., a clause with a single literal) is included to ensure that the entirety of the original formula is true.

Suppose that the input formula F has $\text{vars}(F) = \{x_1, \ldots, x_n\}$. In the output CNF formula, the variables $x_1, \ldots, x_n$ will correspond to the same variables in the input formula. We produce the CNF formula by recursively defining auxiliary variables and emitting clauses for each subformula F' in the input formula as follows.

- If F' is a variable x_i, then the corresponding variable in the CNF formula is also x_i.
- If F' is a negation $\overline{G}$, let x be the literal in the CNF formula corresponding to G; then $\overline{x}$ is the literal in the CNF formula corresponding to F'.
- If F' is a conjunction $G \wedge G'$, let x and y be the literals in the CNF formula corresponding to G and G', respectively; introduce a new variable α corresponding to F'; and ensure that $\alpha \equiv (x \wedge y)$ by emitting the clauses $(\alpha \vee \overline{x} \vee \overline{y})$, $(\overline{\alpha} \vee x)$, and $(\overline{\alpha} \vee y)$.
- If F' is a disjunction $G \vee G'$, let x and y be the literals in the CNF formula corresponding to G and G', respectively; introduce a new variable α corresponding to F'; and ensure that $\alpha \equiv (x \vee y)$ by emitting the clauses $(\overline{\alpha} \vee x \vee y)$, $(\alpha \vee \overline{x})$, and $(\alpha \vee \overline{y})$.
- If F' is a biconditional $G \leftrightarrow G'$, let x and y be the literals in the CNF formula corresponding to G and G', respectively; introduce a new variable α corresponding to F'; and ensure that $\alpha \equiv (x \leftrightarrow y)$ by emitting the clauses $(\alpha \vee x \vee y)$, $(\alpha \vee \overline{x} \vee \overline{y})$, $(\overline{\alpha} \vee x \vee \overline{y})$, and $(\overline{\alpha} \vee \overline{x} \vee y)$.
- If F' is an exclusive disjunction $G \oplus G'$, let x and y be the literals in the CNF formula corresponding to G and G', respectively; introduce a new variable α corresponding to F'; and ensure that $\alpha \equiv (x \oplus y)$ by emitting the clauses $(\overline{\alpha} \vee x \vee y)$, $(\overline{\alpha} \vee \overline{x} \vee \overline{y})$, $(\alpha \vee x \vee \overline{y})$, and $(\alpha \vee \overline{x} \vee y)$.

At the end of this process, we have a variable α corresponding to the entire formula F; emit the unit clause α that requires F to be true.

The rules in the list above can easily be extended to handle additional connectives. For example, if F' is a selection $G\,?\,G' : G''$, whose value is G' if G is true and G'' if G is false, let x, y, and z be the literals in the CNF formula corresponding to G, G', and G'', respectively; introduce a new variable α corresponding to F'; and ensure that $\alpha \equiv (G\,?\,G' : G'')$ by emitting the clauses $(\alpha \vee x \vee \overline{z})$, $(\alpha \vee \overline{x} \vee \overline{y})$, $(\overline{\alpha} \vee \overline{x} \vee y)$, and $(\overline{\alpha} \vee x \vee z)$.

The resulting CNF formula can be reduced in size by maintaining a cache of previously processed subformulas (in some canonical form) and their corresponding variables in the CNF formula. For example, if a new variable α is introduced to correspond to a subformula $G \wedge G'$ in one place in the input formula F, and elsewhere in F the same subformula $G \wedge G'$ appears again, the variable α can be reused.

A further improvement is to identify logically equivalent subformulas. For instance, by De Morgan's laws, a subformula $G \wedge G'$ is logically equivalent to the negation of $\overline{G} \vee \overline{G'}$, so if a variable α has previously been introduced to correspond to the subformula $\overline{G} \vee \overline{G'}$, then $\overline{\alpha}$ can be used to correspond to $G \wedge G'$.

By using this caching strategy and the identification of logically equivalent subformulas, tautologies and contradictions can occasionally be detected. For example, given a conjunction $G \wedge G'$, if α is returned as the CNF literal corresponding to G and $\overline{\alpha}$ is returned as the CNF literal corresponding to G', then the subformula $G \wedge G'$ is a contradiction, and $\bot$ can be returned instead of a variable corresponding to $G \wedge G'$. Similarly, given a disjunction $G \vee G'$, if α is returned as the CNF literal corresponding to G and $\overline{\alpha}$ is returned as the CNF literal corresponding to G', then the subformula $G \vee G'$ is a tautology, and $\top$ can be returned.

These deductions can be propagated up to higher-level clauses as well. For instance, given a conjunction $G \wedge G'$, if the recursive processing of G or of G' returns $\bot$, then $G \wedge G'$ is a contradiction. On the other hand, if the recursive processing of G, say, returns $\top$, then $G \wedge G'$ is logically equivalent to G', so the CNF literal (or the truth value $\top$ or $\bot$) corresponding to G' can be returned directly. Thus there is no need to emit clauses or introduce variables for tautologies or contradictions, and if the input formula F itself is a tautology or a contradiction, then the output CNF formula can be simply $x_1 \vee \overline{x}_1$ or the empty clause Λ, respectively.

6.1.2 Detecting Tseitin clauses in a CNF formula

The Tseitin transformation described above is widely known and commonly used to convert propositional formulas into conjunctive normal form. Many

SAT instances contain subsets of clauses that were produced by this transformation. It is therefore potentially useful to be able to identify these subsets of clauses so as to recover the original propositional formula.

Recall that the standard BDD synthesis algorithm [22, 60, 98] constructs a BDD for a propositional formula by recursively building BDDs for subformulas and then *melding* the BDDs for subformulas F and G to produce a BDD for $F \diamond G$, where $\diamond$ is a Boolean operator such as $\wedge$ or $\oplus$. The motivation for the technique of detecting and extracting Tseitin clauses comes from the observation that this synthesis algorithm often produces efficient BDDs for propositional formulas (i.e., BDDs with few nodes), and if a propositional formula has an efficient BDD representation then the BDD constructed using the top-down algorithms from Chapter 2 may also be of small size.

To identify Tseitin clauses in a SAT instance, we use the following two logical equivalences from the list of rules in Section 6.1.1:

$$[a \equiv (b \wedge c)] \iff (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c),$$
$$[a \equiv (b \leftrightarrow c)] \iff (a \vee b \vee c) \wedge (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee c).$$

Note that we do not need to consider subformulas of the forms $a \equiv (b \vee c)$ and $a \equiv (b \oplus c)$ separately, because $[a \equiv (b \vee c)] \iff [\overline{a} \equiv (\overline{b} \wedge \overline{c})]$ and $[a \equiv (b \oplus c)] \iff [\overline{a} \equiv (b \leftrightarrow \overline{c})]$.

From the equivalences above, we get the following CNF formulas that include the disjunction $x \vee y \vee z$ for literals x, y, and z:

$$[x \equiv (\overline{y} \wedge \overline{z})] \iff (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z}),$$
$$[y \equiv (\overline{x} \wedge \overline{z})] \iff (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{y} \vee \overline{z}),$$
$$[z \equiv (\overline{x} \wedge \overline{y})] \iff (x \vee y \vee z) \wedge (\overline{x} \vee \overline{z}) \wedge (\overline{y} \vee \overline{z}),$$
$$[x \equiv (y \leftrightarrow z)] \iff (x \vee y \vee z) \wedge (x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z).$$

The remaining clauses of the form $a \equiv (b \leftrightarrow c)$ for various combinations of x or $\overline{x}$, y or $\overline{y}$, and z or $\overline{z}$ substituted for a, b, and c in various permutations, whose equivalent CNF formulas contain $x \vee y \vee z$, are all equivalent to the last formula above; for example, $x \equiv (\overline{y} \leftrightarrow \overline{z})$ or $\overline{x} \equiv (y \leftrightarrow \overline{z})$. Also equivalent are $y \equiv (x \leftrightarrow z)$ and $z \equiv (x \leftrightarrow y)$, because these are all equivalent to $x \oplus y \oplus z$. So there is no need to consider them separately.

There is a lot of symmetry in the CNF formula for $x \equiv (y \leftrightarrow z)$. But, considering all combinations of negations of variables, there are only two essentially different formulas of this form: one equivalent to $x \oplus y \oplus z$ and the other equivalent to its negation. The CNF formula for $x \oplus y \oplus z$ includes all disjunctions of three literals with an even number of negations, and the

CNF formula for $\overline{x \oplus y \oplus z}$ includes all disjunctions of three literals with an odd number of negations. So we can canonicalize the forms of these CNF formulas as $|x| \equiv (|y| \leftrightarrow |z|)$ and $\overline{|x|} \equiv (|y| \leftrightarrow |z|)$, respectively, where $|x|$ denotes the positive literal, and $|x| < |y| < |z|$ in a fixed ordering of the variables. We can distinguish between these two canonical forms for a particular disjunction $x \vee y \vee z$ by seeing whether an even or odd number of the literals are negated.

We can identify subsets of clauses in a CNF formula that represent propositional formulas of the form $a \equiv (b \wedge c)$ or $a \equiv (b \leftrightarrow c)$ as follows.

First we iterate through the clauses in the CNF formula and pick out the clauses of length 2. We record these clauses in a hash table (according to the canonical form of the clause) so their existence can be quickly checked. At the same time we pick out the clauses of length 1 (these are fixed variables) and record them in a hash table keyed on the variable, and we pick out the clauses of length 3 and record them in a hash table keyed on the canonical form of the clause (so that we never consider duplicates). We ignore clauses of length 4 or greater.

Next we iterate through the keys of the hash table containing clauses of length 3. Each key is a clause of the form $x \vee y \vee z$, where x, y, and z are literals and $|x| < |y| < |z|$. Therefore the clause is potentially part of a subset of clauses representing any of the following propositional formulas:

- $x \equiv (\overline{y} \wedge \overline{z})$, for which we additionally need $\overline{x} \vee \overline{y}$ and $\overline{x} \vee \overline{z}$;
- $y \equiv (\overline{x} \wedge \overline{z})$, for which we additionally need $\overline{x} \vee \overline{y}$ and $\overline{y} \vee \overline{z}$;
- $z \equiv (\overline{x} \wedge \overline{y})$, for which we additionally need $\overline{x} \vee \overline{z}$ and $\overline{y} \vee \overline{z}$;
- $|x| \equiv (|y| \leftrightarrow |z|)$, if an even number of x, y, and z are negative literals, or $\overline{|x|} \equiv (|y| \leftrightarrow |z|)$, if an odd number of x, y, and z are negative literals.

For the first three possibilities, we check to see whether the two necessary clauses of length 2 exist. If so, we emit all three clauses and any clauses of length 1 for these variables.

For the last possibility, we maintain a hash table whose keys are of the form $a \equiv (b \leftrightarrow c)$. The values in this hash table are numerical counts. When we process a clause that can be part of the representation of $a \equiv (b \leftrightarrow c)$, we increment the corresponding count in this hash table. When a count reaches 4, we know we have seen all of the necessary clauses (because we are not processing duplicate clauses multiple times), so we emit all four clauses and any clauses of length 1 for these variables.

6.2 Graph structure

A SAT instance can be modeled with a graph whose vertices are the clauses. The edges of the graph can be determined and optionally weighted in several ways. For example, the graph may be constructed by joining two vertices with an edge when the corresponding clauses share at least one variable (or literal) in common. Such a graph is called a constraint graph.

The goal of decomposition is to identify and extract a subset of clauses that are in some way tightly dependent on each other. The structure of the constraint graph allows us to model and measure this dependence. If the edges represent pairs of clauses that have at least one variable in common, for example, then a highly connected subgraph corresponds to a subset of clauses each of which shares variables with many others.

For finer control over the graph structure, the threshold for the existence of an edge can be increased to require the clauses to have, say, at least two variables in common. Alternatively, we can weight each edge with the number of common variables in the two corresponding clauses. Such an edge-weighted graph is used in Section 6.3.

Example 6.2.1. Consider the bipartite graph having 10 vertices on each side whose bipartite adjacency matrix is shown below.

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}$$

The first five vertices on the left-hand side (represented by the first five rows in the matrix above) are each joined only to the first four vertices on the right-hand side (represented by the first four columns). The remaining five vertices on the left-hand side are joined to random subsets of four vertices on the right-hand side, so that every vertex on the left-hand side has degree 4.

Because the first five vertices on the left-hand side violate Hall's condition, the bipartite matching problem on this graph is unsatisfiable. A SAT

instance generated from this graph, hall-set-10-4_1.cnf, appears in Appendix A; it is unsatisfiable by construction. This instance has 100 variables and 104 clauses.

The constraint graph for this SAT instance is shown in Figure 6.1, drawn by a force-directed graph drawing algorithm (`neato` from the Graphviz suite[1]). The five black nodes numbered 1 through 5 in the center of the graph represent the first five clauses in the instance, which specify that the first five vertices on the left-hand side must each be joined to one of the first four vertices on the right-hand side. The five light gray nodes numbered 6 through 10, mostly around the perimeter, represent the corresponding constraints for the remaining five vertices on the left-hand side. The 40 dark gray nodes and the 54 white nodes represent the clauses corresponding to triples (v, v', w) of distinct vertices, where v and v' are on the left-hand side and w is on the right-hand side. Such a constraint mandates that the two edges vw and $v'w$ are not both chosen in the matching. The dark gray nodes represent those clauses in which v and v' are among the first five vertices on the left-hand side and w is among the first four vertices on the right-hand side; the white nodes represent the others. Together, the clauses represented by the black nodes and the clauses represented by the dark gray nodes are sufficient to prove the unsatisfiability of the instance; any subinstance including all of these clauses will be unsatisfiable. We refer to the clauses represented by the black and dark gray nodes as the *Hall clauses*.

As can be seen in the figure, the dark gray nodes are generally clustered around the black nodes in the center, as a consequence of the structure of the constraint graph (the force-directed graph drawing algorithm tends to put adjacent nodes near each other and to spread nonadjacent nodes apart). Thus, in this case, the constraint graph structure hints at a useful subinstance. ∎

Unfortunately, graph connectivity alone is not useful in this example for identifying this subinstance, because the nodes corresponding to the Hall clauses do not form a highly connected subgraph of the constraint graph. We encountered similar failures with other instances in our experiments. Additionally, graph connectivity alone seems to be too coarse: the subinstances extracted for various thresholds of connectivity tend to be too large or too small, and it is difficult to find a useful compromise of parameters.

This suggests that we need a more refined measure for the dependencies between clauses of an instance. In particular, it seems that what we need is a way to identify a "neighborhood" of a clause, so that we can extract the

[1] See `http://www.graphviz.org/`.

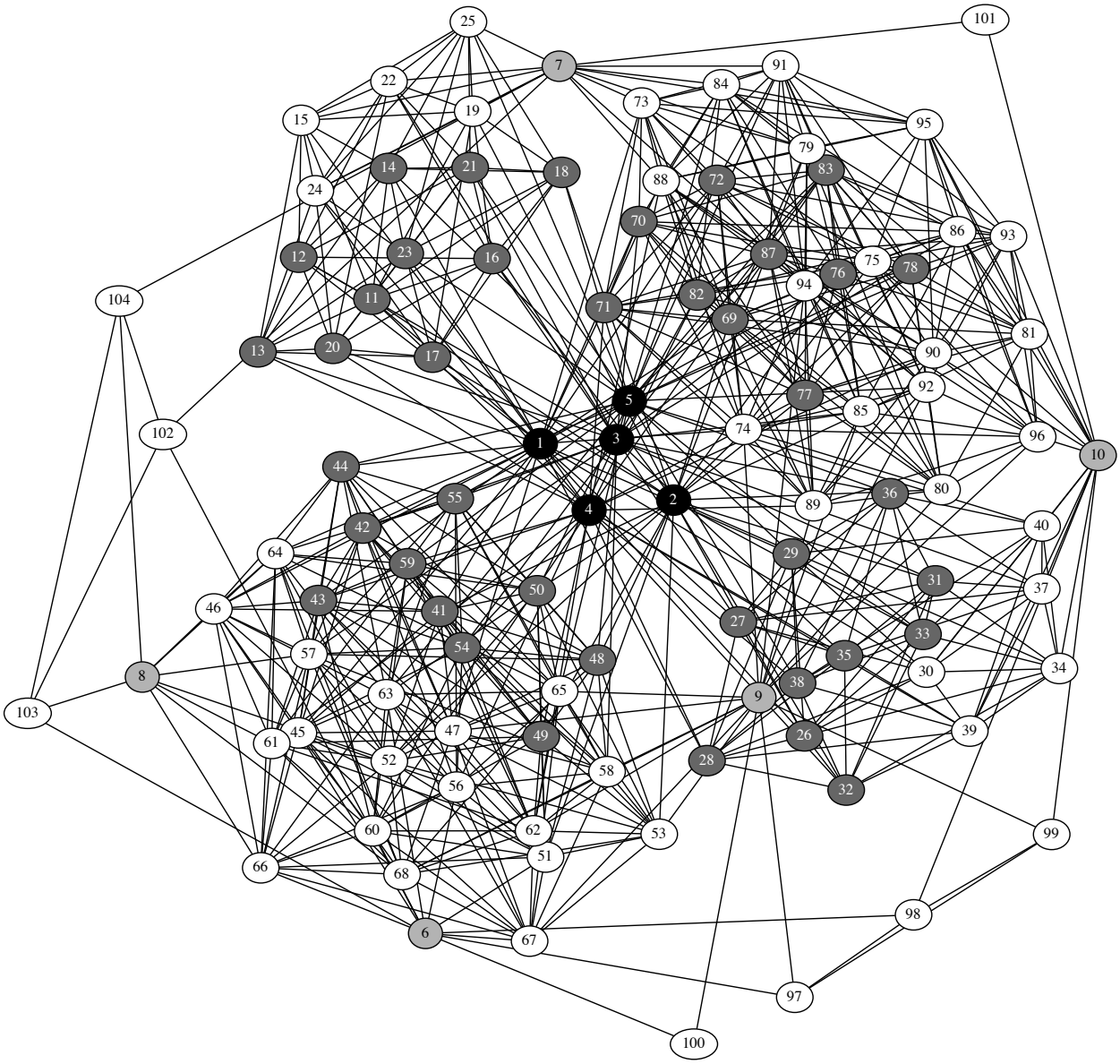

Figure 6.1: The constraint graph for an unsatisfiable bipartite matching instance, drawn by a force-directed graph drawing algorithm.

Figure 6.2: The symbol for a resistor (left) and ground (right).

surrounding neighborhood of a black clause and thereby include all of the dark gray clauses. This idea of identifying a neighborhood of a clause in the constraint graph is the subject of the next section.

6.3 Resistive network decomposition

One way to use the constraint graph structure of a SAT instance to decompose the instance into subinstances is to model the instance as a resistive electrical network. This electrical metaphor has been used before to analyze the structure of graphs, such as to study the behavior of random walks; see Doyle and Snell [33] for an exposition of this application.

A resistor is an element of an electric circuit that impedes the flow of electric current. Resistors are shown in circuit diagrams using the symbol shown on the left in Figure 6.2. The unit of measurement of resistance is the ohm, abbreviated Ω. The reciprocal of resistance is conductance, whose unit of measurement is the mho, abbreviated $\mho$ (also known as the siemens, abbreviated S). The unit of measurement of electric current is the ampere, abbreviated A, and the unit of measurement of voltage is the volt, abbreviated V. The voltage V across a resistor of resistance R carrying a current I is described by Ohm's law, $V = IR$.

To model a SAT instance as a resistive electrical network, we construct a (complete) graph whose nodes are the clauses $C_1, \ldots, C_n$ of the SAT instance. Each edge $\{C_i, C_j\}$ of this graph is treated as a resistor by assigning it a conductance, given by a function $\mho(C_i, C_j)$ of the two clauses corresponding to the endpoints of the edge. An edge joining two "closely related" clauses should be given a large conductance (i.e., a low resistance), so, for example, the conductance of an edge could be the number of variables (or literals) that the two clauses have in common. Note that a conductance of zero is equivalent to a nonexistent edge. Each node C_i is also connected to ground (i.e., a node whose voltage is zero, represented by the symbol on the right in Figure 6.2) by a resistor of conductance $\mho_\bigtriangledown(C_i)$. This could be a constant function, e.g., $\mho_\bigtriangledown(C_i) = 1$, or it could depend on properties of the clause, such as clause length.

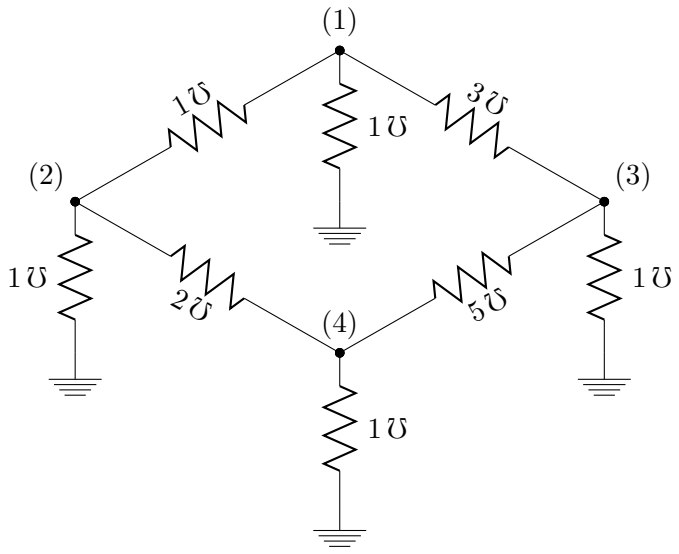

Figure 6.3: The resistive network corresponding to the SAT instance in Example 6.3.1. Conductances are given in mhos.

Example 6.3.1. Consider the following SAT instance.

$$
\begin{array}{ll}
(1) & x_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee x_4 \\
(2) & \overline{x}_1 \vee x_5 \vee \overline{x}_6 \\
(3) & x_2 \vee \overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_7 \vee x_8 \vee \overline{x}_9 \vee x_{10} \vee \overline{x}_{11} \\
(4) & x_5 \vee x_6 \vee x_7 \vee \overline{x}_8 \vee x_9 \vee \overline{x}_{10} \vee x_{11}
\end{array}
$$

Using $\mho_\bigtriangledown(C_i) = 1$ for all i and $\mho(C_i, C_j) = |\mathrm{vars}(C_i) \cap \mathrm{vars}(C_j)|$, that is, the number of variables that the clauses C_i and C_j have in common, we obtain the resistive network shown in Figure 6.3. ∎

Once this resistive network is constructed, we apply a specified voltage to a given node and use Ohm's law to calculate the voltages at all other nodes. The nodes whose voltages exceed a specified threshold give a subset of the clauses of the original SAT instance, and this subset of clauses can be extracted as a subinstance.

The intuitive motivation behind this approach is illustrated in Example 6.2.1: useful subinstances should consist of closely related clauses, and closely related clauses are "nearby" each other in the constraint graph. Thus we should seek a model that can identify "nearby" nodes in a graph.

When voltage is applied to a node of a resistive network of the form described here, it "spreads" to neighboring nodes in accordance with the conductivities between the nodes: nodes joined by a link of high conductance will have very similar voltages, whereas nodes joined by a link of low conductance may have voltages that differ significantly. But the effective conductance between two nodes (and hence their relative voltages) is also affected by the way in which they are mutually connected to other nodes—two nodes that are themselves joined by a link of low conductance may each be connected to a third node by links of high conductance, which provides a high-conductance path between the original two nodes. This meshes well with the notion that two clauses in a SAT instance may not have variables in common themselves, but nevertheless they are related by the fact that they each share several variables with a third clause.

When positive voltage is applied at a node k, then the subset of nodes whose voltages exceed a specified threshold form a *neighborhood* of k. Varying the threshold will of course change the number of nodes in the neighborhood. Since we are identifying nodes in the resistive network with clauses in the SAT instance, we will also speak of the neighborhood of a clause.

If every node is connected to ground via a link with positive conductance, then for any node l having positive voltage there is a path from k to l along which the voltages of the nodes strictly decrease. For there is current flowing from l to ground (under the convention that current flows from positive voltage to ground), and the only source of current in the network is k; so there must be a path from k to l taken by this current, and current flows only from nodes of higher voltage to nodes of lower voltage. Therefore, the neighborhood of a node k always induces a connected subgraph.

A family of neighborhoods of clauses determined in this way is a *full decomposition* of the SAT instance if every clause in the instance is included in at least one of these neighborhoods.

To calculate the voltages at all nodes of the network when voltage is applied at one node, we solve a linear system. For each node i, let V_i be the voltage at i, let $R_{i\triangledown}^{-1} = \mho_{\triangledown}(C_i)$ be the conductance between i and ground, and let $I_{i\triangledown}$ be the current from i to ground. For nodes i and j with $i \neq j$, let $R_{ij}^{-1} = \mho(C_i, C_j)$ be the conductance of the edge $\{i, j\}$, and let I_{ij} be the current flowing from i to j. Note that $I_{ij} = -I_{ji}$. By Ohm's law, $I_{ij} = R_{ij}^{-1}(V_i - V_j)$. (This assumes the convention that that current flows from positive voltage to ground, but this assumption is not materially important—the opposite convention merely results in a sign flip.) Likewise, $I_{i\triangledown} = R_{i\triangledown}^{-1} V_i$. By Kirchhoff's circuit laws, a current conservation property applies to every node (except the one to which voltage is applied): the net

current out of the node must be zero, i.e.,

$$I_{i\triangledown} + \sum_{j \neq i} I_{ij} = 0.$$

(The node to which voltage is applied has a net outflow of current, because the application of voltage acts as a current source; ground acts as a current sink.) By substituting $I_{ij} = R_{ij}^{-1} V_i - R_{ij}^{-1} V_j$ and $I_{i\triangledown} = R_{i\triangledown}^{-1} V_i$ we obtain the equations

$$R_{i\triangledown}^{-1} V_i + \sum_{j \neq i} (R_{ij}^{-1} V_i - R_{ij}^{-1} V_j) = 0.$$

This gives a system of $n-1$ linear equations in n unknowns (V_1 through V_n). The last equation in the system comes from setting V_k to the applied voltage for the node k to which voltage is applied.

It turns out that the system becomes computationally easier to solve if we instead view the process as injecting a certain *current* into the network at the chosen node k, rather than applying a certain voltage. For simplicity, we may assume that the injected current is 1 ampere. Then all of the equations in the system have the same form: for all $i = 1, \ldots, n$,

$$R_{i\triangledown}^{-1} V_i + \sum_{j \neq i} (R_{ij}^{-1} V_i - R_{ij}^{-1} V_j) = \delta_{ik},$$

where k is the index of the chosen node and δ_{ik} is the Kronecker delta, defined by

$$\delta_{ik} = \begin{cases} 1, & \text{if } i = k; \\ 0, & \text{otherwise.} \end{cases}$$

This gives us a symmetric coefficient matrix A. In fact, the following theorem holds:

Theorem 6.3.2. *The coefficient matrix A of the system defined above is positive definite if all conductances to ground $R_{i\triangledown}^{-1}$ are positive, and positive semidefinite if all conductances to ground are nonnegative.*

Proof. The coefficient matrix A has off-diagonal entries $a_{ij} = a_{ji} = -R_{ij}^{-1}$ for $i < j$ and diagonal entries $a_{ii} = R_{i\triangledown}^{-1} + \sum_{j \neq i} R_{ij}^{-1}$. So for any $z \in \mathbb{R}^n$ we

have

$$\begin{aligned} z^{\mathrm{T}} A z &= \underbrace{-2 \sum_{i<j} R_{ij}^{-1} z_i z_j}_{\text{off-diagonal terms}} + \underbrace{\sum_i \left[R_{i\bigtriangledown}^{-1} + \sum_{j \neq i} R_{ij}^{-1} \right] z_i^2}_{\text{diagonal terms}} \\ &= -2 \sum_{i<j} R_{ij}^{-1} z_i z_j + \sum_i R_{i\bigtriangledown}^{-1} z_i^2 + \sum_i \sum_{j \neq i} R_{ij}^{-1} z_i^2 \\ &= \sum_{i<j} R_{ij}^{-1} (z_i^2 + z_j^2 - 2 z_i z_j) + \sum_i R_{i\bigtriangledown}^{-1} z_i^2 \\ &= \sum_{i<j} R_{ij}^{-1} (z_i - z_j)^2 + \sum_i R_{i\bigtriangledown}^{-1} z_i^2. \end{aligned}$$

Both of these summations are nonnegative, and the second is strictly positive for $z \neq 0$ if all conductances to ground $R_{i\bigtriangledown}^{-1}$ are positive. □

As a consequence of Theorem 6.3.2, if all conductances to ground are positive, then the matrix A has a unique Cholesky factorization $A = LL^{\mathrm{T}}$, where L is a lower-triangular matrix with (real) positive entries on the diagonal [49]. Moreover, the matrix A does not depend on which node k is chosen as the current injection point (only the right-hand side of the system changes), so we can compute the Cholesky factorization of A once for a given SAT instance and use it many times to determine neighborhoods of many clauses.

The node voltages obtained by injecting a fixed current (say, 1 ampere) into a given node will not necessarily be the same as the voltages obtained by applying a fixed voltage (say, 1 volt) to that node, but the current-injection solution will be a linear scaling of the voltage-application solution. Therefore, we can obtain the voltage-application solution from the current-injection solution by normalization, namely, by multiplying each node voltage by the reciprocal of the voltage at the node where the current was injected.

Example 6.3.3. Consider the network from Example 6.3.1. Suppose 1 ampere is injected into node 1. Then the system to compute the voltages at each node is

$$\begin{bmatrix} 5 & -1 & -3 & 0 \\ -1 & 4 & 0 & -2 \\ -3 & 0 & 9 & -5 \\ 0 & -2 & -5 & 8 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The Cholesky factorization of the coefficient matrix A above is $A = LL^{\mathrm{T}}$, where

$$L = \begin{bmatrix} \sqrt{5} & 0 & 0 & 0 \\ -1/\sqrt{5} & \sqrt{19/5} & 0 & 0 \\ -3/\sqrt{5} & -3/\sqrt{95} & 3\sqrt{15/19} & 0 \\ 0 & -2\sqrt{5/19} & -101/(3\sqrt{285}) & (\sqrt{401/15})/3 \end{bmatrix}$$
$$\approx \begin{bmatrix} 2.23607 & 0 & 0 & 0 \\ -0.44721 & 1.94936 & 0 & 0 \\ -1.34164 & -0.30779 & 2.66557 & 0 \\ 0 & -1.02598 & -1.99424 & 1.72348 \end{bmatrix}.$$

The solution to this system is

$$\begin{aligned} V_1 &= 152/401 \approx 0.37905, \\ V_2 &= 77/401 \approx 0.19202, \\ V_3 &= 94/401 \approx 0.23441, \\ V_4 &= 78/401 \approx 0.19451. \end{aligned}$$

We normalize these voltages by dividing by V_1 to get

$$\begin{aligned} \hat{V}_1 &= 1, \\ \hat{V}_2 &= 77/152 \approx 0.50658, \\ \hat{V}_3 &= 47/76 \approx 0.61842, \\ \hat{V}_4 &= 39/76 \approx 0.51316. \end{aligned}$$

These are the node voltages that would be obtained by applying 1 volt to node 1. Using these voltages, we can determine a neighborhood of clause 1 by selecting the subset S of clauses whose corresponding node voltages meet or exceed a given threshold voltage V_{th}:

$$S = \{\, C_i : \hat{V}_i \geq V_{\mathrm{th}} \,\}.$$

In this case,

$$S = \begin{cases} \{C_1, C_2, C_3, C_4\}, & \text{if } 0 \leq V_{\mathrm{th}} \leq 77/152; \\ \{C_1, C_3, C_4\}, & \text{if } 77/152 < V_{\mathrm{th}} \leq 39/76; \\ \{C_1, C_3\}, & \text{if } 39/76 < V_{\mathrm{th}} \leq 47/76; \\ \{C_1\}, & \text{if } 47/76 < V_{\mathrm{th}} \leq 1. \end{cases}$$

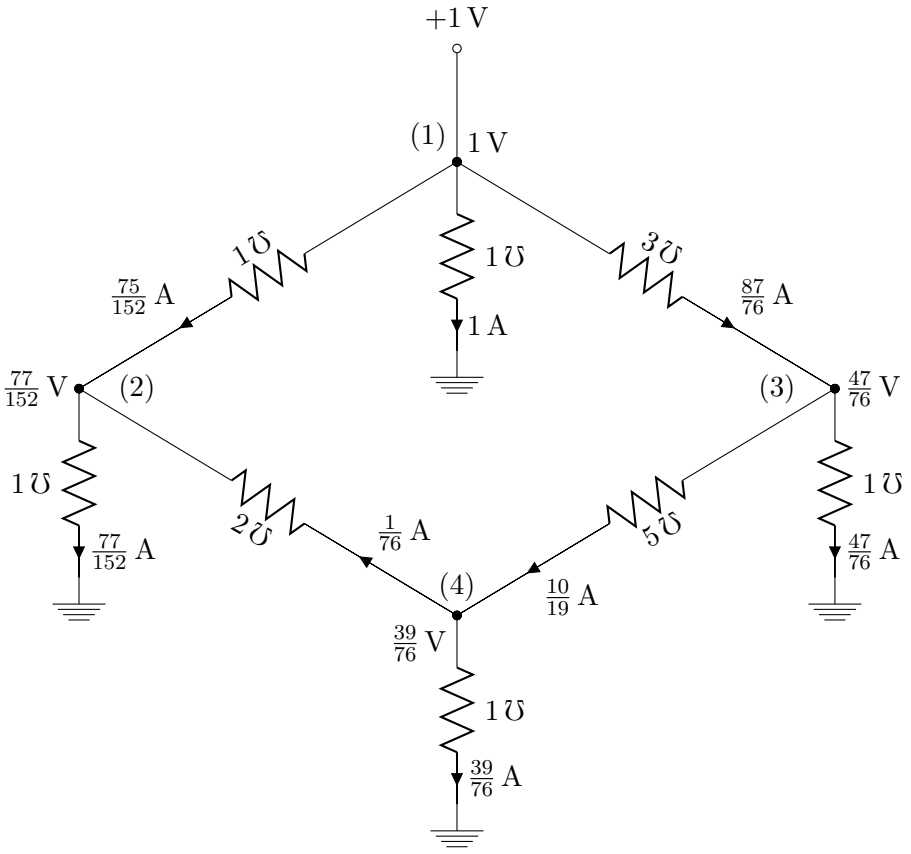

Figure 6.4: Voltages and currents in the resistive network from Figure 6.3 when 1 volt is applied to node 1 (under the convention that current flows from positive voltage to ground).

Note that as the threshold voltage V_{th} is increased from 0 to 1, the size of the neighborhood decreases. This gives us a way to control the size of the extracted subinstance.

The voltages and currents in the network when 1 volt is applied to node 1 are shown in Figure 6.4. ∎

Algorithm 6 uses this resistive network representation of a SAT instance to fully decompose the instance as the union of smaller subinstances. The idea is to choose a node of the network, extract a neighborhood of that node, and repeat this process until every node of the network has been included in at least one of these neighborhoods.

Algorithm 6 Resistive network decomposition of a SAT instance $I = \{C_1, \ldots, C_n\}$

for $i = 1$ **to** n **do**
    $R_{i\triangledown}^{-1} := \mho_{\triangledown}(C_i)$ ▷ conductance to ground
end for
for $i = 1$ **to** $n - 1$ **do**
    for $j = i + 1$ **to** n **do**
        $R_{ij}^{-1} := \mho(C_i, C_j)$ ▷ conductance between clauses
    end for
end for
construct the coefficient matrix A from $\{R_{i\triangledown}^{-1}\}$ and $\{R_{ij}^{-1}\}$
$L :=$ *Cholesky-factorize*(A)
$b := \mathbf{0} \in \mathbb{R}^n$ ▷ right-hand side of system
$U := I$ ▷ clauses that have not yet been covered
while $U \neq \emptyset$ **do**
    select $C_i \in U$
    $b[i] := 1$
    $x :=$ solution to $Ax = b$ using L
    $b[i] := 0$
    $x := x/x[i]$ ▷ normalize
    select threshold voltage V_{th}
    $S := \{\, C_i \in I : x[i] \geq V_{\text{th}} \,\}$ ▷ select subinstance
    output or process S
    $U := U \setminus S$
end while

Several remarks about this algorithm are in order.

The loop on lines 1–3 uses the function $\mho_\bigtriangledown$ to compute a positive conductance to ground for each node, and the loop on lines 4–8 uses the function $\mho$ to compute a nonnegative conductance between each pair of nodes. The behavior of the resistive network can be modified by changing the definitions of these two functions. As a reasonable starting point, we can define $\mho_\bigtriangledown(C_i) = 1$ for all i and $\mho(C_i, C_j) = |\text{vars}(C_i) \cap \text{vars}(C_j)|$, that is, the number of variables that the clauses C_i and C_j have in common.

The set U keeps track of the clauses in the instance that have not yet been included in any subinstance. It is initialized to the entire instance on line 12, and on line 14 a clause C_i is selected from U to act as the current injection point. This clause can be selected in several ways. One possibility is to choose C_i from U uniformly at random. Another possibility is to choose a starting point in the resistive network uniformly at random and then perform a weighted random walk on the network, where the transition probabilities from a given node are proportional to the conductances of the corresponding edges (that is, from node i, the probability of transitioning to node j is $R_{ij}^{-1} / \sum_{k \neq i} R_{ik}^{-1}$). This random walk can be continued for a fixed number of steps, or until an element of U is reached, to choose a clause to act as the current injection point. A fuller exposition of the properties of random walks on electric networks is given by Doyle and Snell [33].

The threshold voltage V_{th} is selected on line 19. The value of V_{th} can be a fixed parameter of the algorithm, or it can be selected dynamically based on the solution x to the linear system $Ax = b$. For example, the mth largest component of x can be selected and this value used for V_{th}; this will yield a subinstance S consisting of (approximately) m clauses (it is possible that V_{th} is the voltage of several nodes, in which case S will contain more than m clauses). This latter approach is useful to decompose a SAT instance into subinstances of (approximately) equal size.

We implemented this technique in C. Our program constructs and solves the linear system to model the injection of current at a clause, using the CHOLMOD package [27] to find the Cholesky decomposition of the coefficient matrix, and outputs a neighborhood of that clause. This can be repeated until the entire instance has been decomposed into subinstances, or it can be applied only to specified clauses. The size of the neighborhoods can be controlled by specifying a number of clauses or a threshold voltage. Optionally, a random walk can be performed, starting at an indicated clause (or at a clause chosen at random), to determine the clause at which current is to be injected.

As a proof of concept, we constructed random unsatisfiable bipartite

matching instances like the one in Example 6.2.1 as follows. A bipartite graph is constructed with n vertices on each side. Each of the first six vertices on the left-hand side is joined only to the first five vertices on the right-hand side. The other vertices on the left-hand side were joined to random subsets of size 5 on the right-hand side. This graph was then converted to an (unsatisfiable) SAT instance modeling the bipartite matching problem.

The SAT instance contains n^2 variables $x_{i,j}$ for $1 \leq i, j \leq n$; the value of the variable $x_{i,j}$ is true if and only if the ith vertex on the left-hand side is matched to the jth vertex on the right-hand side. (The variables $x_{i,j}$ corresponding to edges that do not exist in the graph are ignored; they do not appear in any of the clauses and therefore may take either value.) The clauses are of two types:

- For each vertex v on the left-hand side, there is one clause ensuring that v is matched to some vertex on the right-hand side.
- For each triple (v, v', w) of distinct vertices, where v and v' are on the left-hand side, w is on the right-hand side, and vw and $v'w$ are both edges in the graph, there is one clause ensuring that vw and $v'w$ are not both selected to be in the matching.

(Note that the random structure of the graph means that different instances have slightly different numbers of this second type of clause, even for the same value of n.) The 81 Hall clauses (that is, the clauses corresponding to the subgraph that violates Hall's condition) form an unsatisfiable subset.

We then applied the resistive network decomposition described here to find neighborhoods of a Hall clause of the first type, that is, subsets of clauses whose voltage exceeds a given threshold when a fixed voltage is applied to a clause of the first type within the infeasible Hall set. The number of clauses in the neighborhood was adjusted to the minimum number required (by adjusting the voltage threshold) in order for the neighborhood to include all 81 Hall clauses. This was repeated for 20 different random bipartite graphs for each value of n.

The results, averaged over the 20 trials for each value of n, are shown in Table 6.1. The second and third columns of this table give the number of variables and clauses in the full instance. The last two columns show the size of the neighborhood required to encompass all 81 Hall clauses and the corresponding threshold voltage V_{th}. We see that the neighborhood is relatively small in comparison to the total number of clauses in the instance, demonstrating that the resistive network decomposition is able to effectively identify and extract these closely related clauses.

n	variables	clauses	neighborhood	V_{th}
20	400	320.10	164.85	0.11564
30	900	453.15	171.65	0.10687
40	1600	598.75	192.35	0.09712
50	2500	727.70	173.40	0.09698
75	5625	1072.85	174.55	0.09081
100	10000	1420.45	203.15	0.08625

Table 6.1: Experimental results for resistive network decomposition on random unsatisfiable bipartite matching instances.

6.4 Summary

In this chapter we proposed three techniques for decomposing a SAT instance into smaller subinstances. In Section 6.1 we presented a method for recognizing and extracting subsets of clauses that were produced from Boolean formulas by the Tseitin transformation. Section 6.2 described the constraint graph of a SAT instance and briefly explored the structure of this graph as a means of decomposition. In Section 6.3 we presented a technique to extract neighborhoods of clauses in a SAT instance by modeling the instance as a resistive electrical network, applying voltage at a given clause, and measuring the voltage at the other clauses. Preliminary results show that this method has the potential to identify useful subsets of clauses in the instance, which can be extracted as subinstances.

Chapter 7

Conclusions and outlook

Our aim in this thesis was to develop effective techniques to apply decision diagrams to combinatorial optimization and satisfaction problems, in particular a multidimensional bin packing problem and the Boolean satisfiability problem.

In Chapter 2, we described several variations of a generic algorithm for the construction of exact and approximate decision diagrams representing sets of feasible solutions to constraint satisfaction problems, including a heuristic-driven depth-first method to construct an exact decision diagram and an application of a clustering algorithm to construct approximate decision diagrams.

In Chapter 3, we examined several techniques to work effectively with instances of a multidimensional bin packing problem using MDDs, including the ullage MDD representation to handle symmetry, a rounding-down technique to more reliably detect equivalent nodes, and the identification of free and dead bins to quickly recognize feasibility and infeasibility. Experimental results show that our MDD algorithms, when combined with these representation techniques, can significantly outperform currently used CP techniques and can also consistently outperform MIP.

In Chapter 4, we presented a new algorithm that uses BDDs and resolution to generate valid clauses from a SAT instance. This algorithm can use approximate BDDs for instances that are too large for an exact BDD. We compared the strength of our method to that of SAT conflict analysis and showed that our method can generate strictly stronger clauses than a single application of conflict analysis. Our experimental results show that concatenating these generated clauses to the original instance can significantly reduce the size of the search tree for a SAT solver.

Chapter 5 discusses a few implementation considerations for the practical application of the algorithms described in Chapters 3 and 4.

In Chapter 6, we examined the problem of decomposing a SAT instance into useful subinstances and presented several ideas toward a solution, including the identification and extraction of clauses that were produced by the Tseitin transformation, the decomposition of a SAT instance based on its graph structure, and the modeling of a SAT instance as a resistive electric network in order to identify neighborhoods of clauses.

We conclude with several questions that remain opportunities for future research.

The results from Chapter 4 show that our BDD-guided clause generation method for the SAT problem has a solid theoretical foundation and is potentially stronger than conflict analysis, but finding the best way to put this idea into practice requires more study. Our experimental results show that these clauses can significantly reduce the numbers of conflicts and decisions encountered by the SAT solver during the search. However, the solution time generally does not exhibit a corresponding decrease. There is still much research to be done in discovering heuristics that work well together and in exploring the effects of the number, length, and structure of the generated clauses on the time required by the SAT solver. Preliminary experiments seem to indicate that some subsets of the generated clauses are significantly more helpful than others, which suggests that it may be fruitful to study the properties of these useful subsets and to explore heuristics for selecting such subsets. It may also be productive to adjust the way in which the SAT solver uses the generated clauses (for example, by treating them in the same way as its own learned clauses rather than as clauses in the original instance). Alternatively, rather than performing the clause generation as a preprocessing step, the generation of witness clauses may be more closely integrated into the SAT solver. For instance, when the solver appears to be making very little progress, and the number of free variables is limited, we can interrupt the search and give the rest of the formula to a BDD-based algorithm to generate witness clauses conditional on the partial assignment represented by the last search state.

The decomposition techniques discussed in Chapter 6 are also deserving of further investigation. Only preliminary experiments have been done so far. In particular, can the resistive network decomposition technique be effectively applied to large real-world SAT instances to identify subinstances from which useful clauses can be derived? The graph-theoretical properties of the neighborhoods obtained using this approach are also worth investigating.

Propositional model counting, or #SAT, is the problem of determining the number of satisfying solutions to a SAT instance [43]. Areas of study in which model counting is useful include the analysis of combinatorial problems, such as combinatorial designs, and various probabilistic inference problems, such as Bayesian net reasoning. This is a challenging problem; it is #P-complete, and current model counters are not scalable for many problems. Because model counters must perform exhaustive analyses for both satisfiable and unsatisfiable instances, nogoods are especially beneficial. Consequently, it may be profitable to extend the application of our witness-clause method to improve the performance of model counters.

Much of the existing theory of BDDs (see Knuth [60], for example) is based on a fundamental synthesis algorithm that takes the BDDs for two Boolean functions f and g and produces the BDD for the function $f \diamond g$, where $\diamond$ is a Boolean operator; this can be viewed as a "bottom-up" process for BDD construction and results in reduced BDDs. However, the construction algorithms that we have presented and applied in this thesis use a different, top-down approach that does not always produce structurally reduced decision diagrams. It would be enlightening to extend or adapt the existing theory to these top-down algorithms. The theory of approximate decision diagrams is also relatively unexplored and awaits results such as provable bounds on the effectiveness of approximate decision diagrams for various problems.

Appendix A

Experimental instances

In this appendix we present some of the instances that were used in the experiments reported in Sections 3.5, 4.4, and 6.2. These instances are also available in machine-readable form by request.

A.1 Multidimensional bin packing instances

Each of the following multidimensional bin packing instances has 6 dimensions, 18 items, and 6 identical bins. The columns of the table correspond to the 6 dimensions. The first 18 rows of the table specify the sizes of the items, and the last row specifies the size of each of the bins (every bin has the same 6-dimensional size). These are the 52 instances with 20% bin slack examined in detail in Section 3.5. The full set of multidimensional bin packing instances used in the experiments in Section 3.5, having bin slack between 0% and 35%, is available at `http://www.math.cmu.edu/~bkell/6-18-6-instances.txt` or by request.

907	356	772	517	721	511
403	916	467	470	678	666
611	791	111	47	82	304
847	771	778	432	798	17
966	224	90	312	339	765
407	961	105	677	298	868
472	845	973	844	547	144
806	453	572	487	211	608
729	87	363	94	958	483
193	203	380	841	731	178
184	789	957	568	565	725
321	761	744	72	377	963
818	543	475	809	725	379
719	903	353	279	469	791
704	690	610	95	506	46
632	181	351	941	570	242
770	482	258	408	472	308
953	598	78	524	860	188
2289	2111	1688	1684	1982	1638

801	925	825	138	791	510
75	125	465	26	610	103
279	827	681	871	819	22
855	171	953	709	108	506
299	404	328	639	313	240
931	483	557	466	476	728
784	704	591	625	698	584
318	620	574	172	863	226
252	581	62	310	330	383
433	247	716	806	825	577
617	258	238	445	95	993
758	489	697	157	804	562
635	381	336	486	23	989
848	742	333	99	568	400
135	490	730	646	104	533
957	212	636	439	3	679
554	185	336	894	689	685
345	976	917	267	311	251
1976	1764	1995	1639	1686	1795

435	147	454	285	822	495
660	409	353	180	33	903
618	293	493	529	791	455
847	342	252	882	860	966
449	928	91	477	752	5
981	591	298	762	860	691
725	696	967	82	487	647
228	861	191	377	904	765
369	191	585	645	254	838
676	434	277	791	859	218
877	174	680	440	986	759
475	107	881	566	807	60
713	434	270	727	949	559
444	90	510	320	477	308
133	836	337	78	70	145
97	372	441	749	907	190
971	348	287	797	466	943
395	586	34	750	595	188
2019	1568	1481	1888	2376	1827

184	846	679	19	580	429
496	180	594	579	428	287
166	267	905	629	967	684
322	713	140	844	448	464
439	617	461	640	383	718
19	501	485	682	289	319
174	970	676	145	881	364
142	946	850	588	298	265
166	875	613	557	863	590
702	162	664	475	999	903
147	125	35	347	113	857
984	123	559	560	597	927
540	47	197	21	347	208
43	947	287	722	710	333
343	122	441	788	289	796
775	737	861	225	674	38
528	381	691	380	369	594
201	58	806	247	208	870
1275	1724	1989	1690	1889	1930

622	195	984	735	87	649
710	978	213	274	209	754
878	312	49	160	482	904
416	85	126	252	45	205
88	944	536	896	489	823
38	106	888	837	239	665
987	612	670	621	457	813
76	296	398	390	145	143
25	828	801	905	618	395
825	516	530	689	435	335
398	187	21	213	417	962
279	739	943	219	582	960
861	878	394	894	525	810
417	994	992	943	367	5
232	872	859	803	430	415
211	382	996	905	823	111
816	361	766	666	568	457
393	167	189	937	608	327
1655	1891	2071	2268	1506	1947

212	719	641	998	784	790
178	94	37	414	594	838
707	113	806	149	264	829
819	541	320	207	22	775
931	24	43	628	496	396
948	264	733	555	296	834
854	891	110	381	14	8
346	210	792	987	133	642
483	197	388	107	178	573
595	485	496	349	723	401
162	826	107	730	667	855
707	45	617	664	533	372
155	432	512	578	180	115
968	707	164	324	523	751
806	787	573	90	77	735
731	276	728	166	30	560
134	158	530	386	214	343
415	598	609	716	565	574
2031	1474	1642	1686	1259	2079

670	604	911	675	419	923
294	750	275	344	991	354
391	768	305	82	265	199
366	303	740	104	397	388
609	130	231	59	185	490
748	331	422	148	603	971
451	340	324	634	697	391
600	804	569	402	28	259
537	49	264	380	625	192
777	766	826	80	969	107
346	102	573	4	211	947
624	537	750	360	894	641
347	921	966	669	858	374
805	871	703	999	573	616
52	960	588	266	856	989
659	470	205	30	731	681
869	334	198	777	831	69
733	162	820	730	478	523
1976	1841	1934	1349	2123	1823

469	753	354	325	98	717
575	899	593	585	696	563
720	808	156	50	697	666
344	633	335	567	14	633
552	379	352	506	133	112
409	320	454	711	180	144
371	69	135	268	221	704
40	46	536	816	371	947
964	463	760	825	387	75
946	746	387	947	541	223
669	57	919	89	155	661
366	625	74	291	395	192
676	3	4	872	66	371
914	532	344	801	495	387
94	464	581	936	607	781
374	541	68	782	234	965
746	322	546	660	982	221
901	594	365	453	9	782
2026	1651	1393	2097	1257	1829

43	932	988	781	476	244
167	730	371	41	823	813
205	260	527	836	556	25
395	756	218	98	756	258
843	969	626	271	987	181
245	849	666	346	211	430
834	818	789	154	437	83
226	485	163	572	899	511
909	405	405	34	130	994
497	651	523	357	664	815
157	938	263	981	554	504
64	816	714	788	239	410
563	746	233	239	812	229
219	865	168	767	281	364
131	428	373	860	612	628
623	188	344	753	205	112
360	476	373	178	222	960
157	863	61	522	200	531
1328	2435	1561	1716	1813	1619

336	918	321	230	359	233
756	358	621	949	372	552
56	837	438	626	702	627
748	155	92	700	950	333
181	978	338	977	428	687
942	884	961	159	791	125
418	509	568	440	6	201
530	215	90	498	635	970
509	948	82	218	35	110
204	641	45	744	759	255
301	143	969	34	877	683
902	305	150	403	415	289
397	202	563	665	887	661
491	271	473	506	505	230
467	696	728	114	355	507
847	863	898	950	169	76
934	817	99	903	951	770
54	725	393	979	268	153
1815	2093	1566	2019	1893	1493

67	885	48	539	184	164
213	130	597	45	248	87
799	822	250	117	878	588
404	15	294	519	343	711
218	57	564	769	112	543
664	779	957	670	217	349
117	136	315	767	995	30
787	856	327	117	792	937
694	792	877	195	971	241
731	633	99	835	919	884
631	595	864	918	990	751
253	157	146	317	286	714
362	285	960	858	174	634
697	752	607	676	974	401
880	688	772	276	262	939
433	825	551	677	107	459
281	513	935	42	910	434
1838	1973	1984	1761	1898	1901

552	857	403	173	281	214
350	521	540	896	534	236
804	766	576	594	490	663
146	603	70	26	198	472
567	786	95	560	915	765
574	196	967	405	440	674
30	408	272	107	420	667
504	359	223	17	416	34
298	456	391	918	874	939
840	109	270	137	906	909
836	852	485	592	625	600
341	516	643	630	107	427
793	424	382	553	701	359
442	636	284	931	437	962
703	347	708	70	37	395
433	529	63	232	505	324
834	22	301	66	576	453
259	700	393	250	633	551
1862	1818	1414	1432	1819	1929

803	243	79	538	14	849
589	324	802	908	310	767
718	690	414	642	489	613
347	424	144	626	55	663
563	283	769	24	214	266
835	950	935	724	244	512
493	819	472	823	919	711
29	691	853	684	289	842
729	362	783	772	497	18
113	392	725	305	10	442
97	202	363	747	555	259
150	638	751	481	403	562
241	203	124	373	827	514
929	651	275	713	504	82
357	954	328	950	108	550
468	472	169	710	498	971
538	974	511	120	73	931
918	874	660	155	802	65
1784	2030	1832	2059	1363	1924

99	539	893	138	163	857
899	359	227	550	933	890
277	88	955	19	910	136
375	418	815	802	994	935
613	248	698	301	89	547
515	539	619	99	491	586
951	576	299	926	30	32
433	146	542	987	481	22
395	892	298	747	956	718
625	59	407	401	658	155
683	210	268	513	296	305
17	707	661	406	786	501
626	936	221	135	888	637
53	41	519	628	985	627
920	343	372	556	478	781
803	983	770	606	413	964
956	317	309	560	813	832
398	334	742	353	568	802
1928	1547	1923	1746	2187	2066

517	730	152	361	515	601
80	687	910	807	0	942
994	895	238	797	413	898
109	557	431	219	22	141
178	321	382	47	853	377
447	785	951	201	952	319
652	927	895	939	240	306
99	763	899	92	766	342
718	236	633	888	941	358
142	847	380	147	309	716
399	351	313	320	119	614
463	682	770	589	483	421
463	797	32	443	760	156
54	889	422	289	377	785
219	365	454	775	696	624
547	925	512	762	286	426
681	865	124	590	183	716
403	23	703	788	530	885
1433	2329	1841	1811	1689	1926

192	961	569	264	727	859
383	112	150	76	344	825
135	351	145	809	998	13
679	179	724	332	594	812
354	126	162	650	446	326
632	267	149	21	718	152
974	730	704	129	56	732
530	14	331	709	349	499
93	403	354	609	367	648
532	969	151	547	824	211
25	807	171	608	62	904
282	308	885	50	205	315
299	228	997	89	988	126
596	615	453	541	972	18
987	289	937	565	211	2
203	954	977	104	773	299
843	406	330	725	790	439
771	252	910	967	442	372
1702	1595	1820	1559	1974	1511

837	944	825	897	416	696
603	926	688	627	489	532
631	858	200	417	770	492
286	29	361	304	383	734
80	983	636	619	230	184
694	346	615	660	188	175
763	633	690	477	443	806
413	229	25	76	272	403
66	378	911	477	37	29
533	545	136	66	794	270
409	349	776	943	968	566
23	755	202	421	234	994
930	67	845	491	586	784
483	841	645	926	605	285
814	320	338	701	443	208
339	936	60	901	265	958
244	147	277	987	36	366
352	911	548	386	780	391
1700	2040	1756	2076	1588	1775

963	482	311	993	711	12
197	298	585	95	314	545
917	248	338	985	162	905
495	781	858	995	987	383
561	151	358	451	381	722
890	304	637	605	505	576
51	444	536	250	60	269
371	997	301	395	681	7
739	210	288	630	254	506
571	301	352	134	303	783
610	632	408	525	570	351
28	604	36	226	122	480
360	637	96	592	530	82
762	828	667	254	598	656
868	725	681	610	990	965
571	34	448	271	106	521
585	212	181	330	951	815
903	887	920	817	707	531
2089	1755	1601	1832	1787	1822

485	303	704	783	475	249
643	802	919	141	672	15
544	195	442	249	270	682
471	234	177	354	365	232
394	867	320	835	523	454
724	799	682	556	492	730
628	728	260	166	54	756
693	856	327	242	802	670
390	75	259	217	561	100
924	767	82	787	239	487
21	285	692	768	705	652
437	251	743	115	165	127
681	276	95	281	201	916
243	721	723	294	293	385
459	506	123	729	936	173
164	125	479	632	436	176
671	5	191	996	599	375
861	290	560	252	647	246
1887	1617	1556	1680	1687	1485

113	834	608	136	151	396
457	417	729	117	51	313
143	311	474	102	350	660
555	759	807	149	244	475
999	415	900	566	678	381
368	535	510	93	611	286
970	120	431	155	615	611
145	685	264	669	660	241
989	744	988	925	829	936
24	727	133	176	729	742
368	334	406	111	233	582
604	905	141	955	398	885
730	929	755	397	966	18
462	431	555	50	397	688
689	993	404	639	602	690
246	107	54	810	157	936
910	615	911	165	82	843
761	106	428	360	736	636
1907	1994	1900	1315	1698	2064

555	168	336	318	376	814
543	488	224	383	116	480
396	963	965	369	347	258
426	449	656	854	87	933
296	418	377	154	163	193
97	254	869	384	795	395
498	313	164	834	640	962
18	982	317	235	575	564
941	907	46	683	434	270
525	134	459	89	97	742
912	534	539	760	828	451
647	325	101	262	676	738
412	555	266	924	894	814
817	715	613	312	597	967
816	726	924	545	687	114
680	718	439	917	444	284
998	582	48	70	905	807
628	656	392	376	91	862
2041	1978	1547	1694	1751	2130

574	800	506	945	631	779
759	974	208	261	902	843
379	968	99	645	756	852
992	54	289	422	613	481
190	370	766	475	611	381
633	620	514	960	320	89
40	401	21	800	579	1
656	625	569	410	635	605
741	821	756	124	747	178
985	972	227	15	697	131
26	244	359	821	774	628
387	713	309	298	277	217
31	435	867	131	45	427
392	440	927	540	671	889
519	808	677	22	546	312
350	893	739	23	122	424
980	954	376	24	845	738
210	775	712	297	278	862
1769	2374	1785	1443	2010	1768

384	487	480	837	240	407
309	770	817	428	803	249
281	578	448	655	436	18
978	881	827	770	968	271
129	137	861	440	613	790
696	415	134	574	359	95
902	919	2	392	452	132
442	595	423	720	620	872
288	535	643	498	471	926
472	955	15	12	573	756
975	210	70	182	813	243
262	965	59	357	339	776
722	398	509	73	792	144
373	341	922	636	750	231
972	920	157	259	302	201
614	438	681	877	960	776
424	530	794	84	853	592
116	153	198	304	486	677
1868	2046	1608	1620	2166	1632

259	677	979	906	323	741
793	553	478	258	489	453
497	174	231	271	311	544
912	267	523	398	73	953
540	244	375	882	330	960
488	94	413	158	250	265
693	457	609	772	364	875
631	24	482	117	710	740
712	638	61	476	251	616
85	980	334	201	945	584
811	130	271	818	739	650
996	273	355	831	350	440
433	810	495	704	157	579
131	604	835	953	330	455
1	32	318	741	332	555
397	390	80	482	432	500
594	304	538	533	415	369
483	115	204	120	663	459
1892	1354	1517	1925	1493	2148

366	674	839	194	628	309
268	563	493	626	771	830
490	128	512	41	103	679
830	501	738	210	758	347
656	423	187	591	220	253
955	550	932	115	507	3
572	622	512	491	136	761
578	815	539	638	92	962
6	763	417	73	441	387
120	558	442	879	748	236
646	203	450	254	629	918
165	760	6	202	933	56
416	846	669	453	287	419
346	715	719	202	373	750
433	761	721	932	709	81
605	287	517	937	507	21
362	479	521	677	699	710
295	531	173	305	60	722
1622	2036	1878	1564	1721	1689

26	984	175	937	341	524
107	277	461	541	249	234
585	943	64	521	262	982
335	376	844	55	539	393
794	838	38	119	771	325
296	994	75	851	940	672
641	186	235	571	761	112
906	67	304	862	525	95
863	293	315	240	980	90
462	110	235	33	149	893
959	870	579	564	667	365
560	459	884	908	327	906
5	439	744	968	162	296
808	370	246	355	892	372
124	952	56	726	290	18
727	466	246	193	550	748
610	841	495	919	69	413
143	79	690	580	536	375
1791	1909	1338	1989	1802	1563

269	261	798	845	512	176
994	564	97	906	98	502
617	348	290	841	507	215
361	651	155	831	133	69
588	300	580	793	825	517
926	72	723	688	422	722
847	979	383	558	127	564
372	307	815	747	8	349
686	471	445	610	500	464
259	742	672	901	646	771
941	38	648	327	444	800
975	436	542	535	167	38
214	587	825	816	548	92
627	284	169	585	720	814
433	553	940	402	793	439
970	210	422	489	389	510
101	652	562	489	355	152
218	134	546	246	413	119
2080	1518	1923	2322	1522	1463

559	0	611	834	932	980
96	691	456	1000	837	840
909	913	89	602	870	936
790	647	699	389	777	251
441	423	199	669	333	320
401	612	975	671	558	387
286	914	309	624	111	293
614	195	631	256	65	728
435	283	69	202	332	742
188	13	61	568	429	614
820	708	617	12	524	306
239	525	256	417	419	938
510	341	856	818	980	109
869	403	404	498	578	436
570	738	253	11	543	412
705	284	931	308	20	921
835	988	648	664	313	303
248	391	626	595	435	704
1903	1814	1738	1828	1812	2044

672	111	99	639	826	960
605	569	532	727	418	372
49	475	789	190	910	502
7	177	920	137	568	484
810	956	33	467	454	958
83	121	803	785	159	931
454	633	66	266	790	222
402	300	389	427	262	284
379	940	530	829	515	39
870	55	717	927	520	997
488	463	790	318	960	921
423	88	508	871	375	329
497	298	458	559	755	389
227	411	252	798	255	711
226	559	271	746	713	616
129	371	886	222	150	37
438	588	691	678	643	27
852	603	904	334	383	16
1523	1544	1928	1984	1932	1759

435	788	335	522	191	290
808	618	344	181	658	70
299	11	376	169	568	822
547	562	53	575	156	756
3	510	38	296	362	999
386	639	849	312	613	610
287	744	285	35	963	196
644	686	156	922	44	45
840	423	199	516	968	384
829	74	200	872	899	886
306	907	602	705	802	118
849	43	265	10	927	442
772	226	813	324	498	529
654	802	128	402	271	670
918	819	936	270	861	945
414	986	336	80	717	34
360	531	580	393	916	934
168	70	170	925	504	458
1904	1888	1333	1502	2184	1838

54	523	364	109	936	517
61	342	954	263	357	582
381	244	375	967	17	31
153	727	734	537	997	214
834	821	957	464	578	303
839	668	545	593	758	183
233	70	121	260	202	920
635	726	675	209	938	527
745	890	38	718	485	736
107	802	572	665	921	446
164	800	9	32	307	607
539	361	407	727	881	432
14	358	921	842	981	933
432	668	960	265	340	358
989	177	155	786	799	744
174	781	21	93	75	574
610	257	534	753	705	447
835	329	932	197	43	555
1560	1909	1855	1696	2064	1822

476	128	585	720	782	48
470	526	307	343	525	550
344	514	385	376	490	886
244	603	432	835	444	897
969	995	488	591	694	471
75	422	348	456	200	494
367	906	388	870	124	921
52	505	711	910	145	936
815	228	663	417	777	634
864	115	139	928	902	423
654	639	386	838	132	4
603	834	855	980	871	516
719	279	551	601	669	36
729	68	205	493	553	581
73	978	480	631	205	297
253	276	236	968	388	725
611	202	427	936	517	820
539	879	272	893	455	55
1772	1820	1572	2558	1775	1859

665	478	342	397	559	876
288	576	375	624	642	848
148	227	228	429	150	796
57	900	420	288	837	811
317	293	995	966	410	868
634	470	259	183	196	836
774	404	842	281	337	999
522	84	794	907	513	311
165	218	71	133	352	814
47	683	23	555	374	549
900	495	552	158	560	133
735	480	262	227	810	727
868	506	272	660	919	420
337	484	514	63	913	732
465	184	764	738	910	175
799	372	637	58	900	954
771	939	58	255	439	146
388	497	297	57	192	321
1776	1658	1541	1396	2003	2264

506	966	272	77	3	133
288	998	428	312	869	373
162	15	183	94	683	903
266	114	708	692	210	613
679	138	220	949	151	782
930	187	452	629	554	660
237	448	807	388	459	476
258	899	809	199	373	211
977	389	352	215	731	903
745	40	857	759	558	211
512	990	344	44	50	51
160	548	523	125	442	244
562	15	601	855	584	732
952	125	660	575	205	615
122	872	380	265	558	599
394	882	654	371	407	278
347	586	206	111	72	683
100	514	608	25	182	103
1640	1746	1813	1337	1419	1714

201	306	730	826	953	14
253	812	527	783	650	531
72	951	655	877	731	792
850	685	943	472	61	701
421	172	111	571	754	62
794	900	400	832	417	630
162	403	534	68	374	874
265	55	603	759	691	899
82	369	816	819	132	419
0	446	639	712	797	421
666	315	569	726	494	700
608	389	336	789	598	878
532	416	974	636	196	556
772	330	297	216	875	248
839	51	788	446	413	712
393	162	993	920	624	183
422	845	869	513	996	780
152	184	921	814	613	892
1497	1559	2341	2356	2074	2059

1	553	445	376	299	921
941	729	405	284	525	600
750	835	453	209	588	765
213	941	570	313	941	638
585	723	257	513	109	642
250	204	326	874	88	753
107	868	819	719	881	314
428	491	543	42	557	417
845	267	334	612	381	793
673	543	696	224	122	66
629	750	613	602	426	896
583	23	783	861	998	295
397	624	816	276	372	544
319	961	849	256	845	869
279	192	947	80	214	989
456	697	703	512	687	873
440	654	965	927	876	332
559	949	449	47	376	611
1691	2201	2195	1546	1857	2264

284	57	755	294	543	672
791	975	298	12	34	203
208	718	679	107	953	930
319	218	614	300	709	856
300	36	619	600	750	198
160	814	890	463	420	87
664	435	55	448	174	231
226	628	261	773	493	959
977	644	789	507	995	775
476	216	934	86	902	311
910	142	779	268	890	873
798	212	311	877	940	756
280	736	41	718	940	2
368	288	43	83	66	904
355	939	79	856	139	94
762	617	90	974	946	74
395	238	650	634	857	196
435	279	979	439	567	457
1742	1639	1774	1688	2264	1716

167	818	608	814	23	548
961	39	870	415	511	737
236	113	691	741	439	313
751	450	897	956	788	622
833	446	733	891	631	427
358	585	372	464	367	650
250	395	230	153	680	283
991	294	781	1000	696	307
350	899	249	948	611	936
661	856	398	572	113	59
538	847	764	733	869	199
611	740	521	266	524	201
675	824	608	686	854	439
293	22	954	15	469	209
610	889	159	462	268	398
421	117	56	276	72	918
775	654	31	15	658	678
832	811	989	131	221	318
2063	1960	1983	1908	1759	1649

852	507	72	408	14	558
278	244	897	702	823	546
607	595	51	55	602	718
337	142	711	707	156	843
784	437	826	399	685	941
111	975	30	649	934	304
333	639	115	394	890	519
751	49	931	414	281	103
456	88	255	594	99	790
838	876	626	93	303	200
901	73	156	610	224	390
558	654	89	802	509	911
740	257	925	800	318	268
721	620	102	475	547	932
178	102	948	999	653	457
635	174	464	253	654	542
782	161	344	969	972	955
986	524	570	844	752	946
2170	1424	1623	2034	1884	2185

469	238	374	848	904	960
887	446	204	110	151	270
699	229	575	122	973	79
266	764	719	627	732	843
540	333	412	940	728	347
950	576	73	936	273	187
74	696	641	890	944	802
644	181	770	24	225	390
550	376	336	629	919	406
999	613	249	185	392	640
267	959	664	981	236	875
503	547	86	477	920	979
539	593	914	934	159	245
726	367	748	710	977	216
173	669	354	480	260	735
568	862	169	650	550	643
79	42	684	6	108	176
357	517	403	153	790	953
1858	1802	1675	1941	2049	1950

845	989	248	81	305	972
883	569	237	518	139	489
225	117	518	970	600	457
872	505	599	840	257	752
698	281	923	737	956	544
648	65	321	890	396	996
485	565	774	563	434	497
868	647	95	799	442	240
564	537	358	861	728	695
469	902	345	92	255	79
356	281	887	342	164	405
406	256	565	711	71	717
202	565	492	116	143	758
900	10	208	467	252	845
346	752	513	660	681	794
951	262	880	505	497	716
528	163	519	413	481	410
660	475	758	504	647	148
2182	1589	1848	2014	1490	2103

522	94	826	245	623	359
475	769	166	439	589	487
88	130	255	578	393	358
75	532	220	579	818	456
395	21	415	630	369	84
858	940	171	438	777	870
91	354	0	308	705	865
457	800	28	341	0	765
272	735	406	844	908	722
946	941	776	707	183	31
237	613	98	684	910	563
872	908	710	262	701	914
637	862	960	825	780	231
431	392	466	518	268	738
249	987	351	921	417	762
60	338	61	500	157	596
136	103	344	636	916	171
207	201	706	160	274	675
1402	1944	1392	1923	1958	1930

870	223	637	982	142	395
442	992	320	47	900	162
952	281	403	642	276	289
92	441	583	536	949	116
909	56	719	90	648	862
613	5	435	891	209	111
856	324	751	425	258	83
788	627	208	980	437	989
450	462	7	78	514	554
668	629	856	813	282	497
350	263	950	293	38	717
569	354	22	26	588	659
773	392	560	37	166	590
75	867	186	52	585	497
438	715	155	368	476	787
624	726	417	640	811	977
863	917	889	370	611	526
549	966	411	674	173	969
2177	1848	1702	1589	1613	1956

563	155	777	198	257	801
475	24	142	52	257	146
926	908	377	355	431	99
226	706	72	722	548	507
41	346	131	391	795	752
56	253	16	865	920	928
340	364	40	302	246	932
760	442	153	818	53	522
892	63	349	648	417	482
130	197	600	357	253	956
603	112	383	894	4	694
630	497	448	861	901	887
822	695	477	4	106	197
809	357	652	295	372	336
231	889	653	931	827	734
697	221	230	57	373	432
661	466	75	894	592	268
658	828	389	405	523	124
1904	1505	1193	1810	1575	1960

70	767	387	291	65	307
881	62	755	409	654	536
414	99	797	555	652	157
563	525	134	762	842	65
280	438	452	974	440	923
203	603	123	823	577	247
484	73	829	117	700	617
408	21	522	476	870	586
708	401	550	307	654	229
672	542	374	539	746	10
455	379	358	85	585	408
610	624	365	310	130	881
142	183	495	669	462	589
178	967	890	327	946	797
513	230	452	896	974	353
767	219	391	951	76	659
435	468	61	312	978	993
676	701	862	683	945	548
1692	1461	1760	1898	2260	1781

766	313	798	401	807	719
149	177	26	678	443	540
312	842	554	754	923	628
473	786	238	983	168	700
739	831	738	885	892	937
800	147	40	971	587	150
971	379	615	304	791	660
354	76	180	516	804	933
70	142	218	567	941	346
333	23	933	180	392	705
510	111	438	514	565	289
191	282	161	286	37	707
467	340	136	117	464	440
601	961	967	855	914	486
873	136	669	349	581	755
269	938	879	554	906	463
913	503	579	247	886	143
890	353	691	571	48	688
1937	1468	1772	1947	2230	2058

946	165	344	812	951	433
730	917	280	67	286	151
983	744	117	29	926	286
21	220	401	518	639	971
193	921	550	105	756	767
554	334	181	494	972	713
129	675	918	655	768	78
344	511	660	12	110	352
709	230	70	628	854	119
978	723	304	753	80	5
174	613	190	248	810	26
639	528	529	565	825	463
222	970	542	868	832	852
125	613	810	877	895	420
173	75	443	645	5	796
537	433	38	452	887	321
755	536	536	378	88	611
29	238	87	669	518	550
1649	1890	1400	1755	2241	1583

452	856	880	851	948	856
964	831	604	137	562	733
6	29	982	405	567	647
180	243	102	19	225	710
342	522	75	986	317	424
234	134	782	962	467	132
887	569	658	728	438	136
14	115	689	771	912	283
764	29	987	626	245	498
406	547	95	49	375	470
823	142	480	942	959	412
571	38	401	528	813	186
420	745	740	684	756	653
812	441	310	321	955	56
91	502	56	334	902	762
260	625	305	862	447	613
597	150	605	1000	293	28
979	277	818	428	501	24
1761	1359	1914	2127	2137	1525

256	167	271	375	927	86
405	811	469	118	795	606
970	603	343	620	526	394
507	676	811	479	482	439
388	592	868	3	189	633
57	926	313	811	810	979
180	166	975	518	563	866
931	213	116	743	442	798
346	736	354	132	784	362
283	98	275	675	307	861
240	461	290	895	711	801
514	326	875	651	863	16
782	164	114	77	496	30
72	792	522	540	346	647
471	798	269	777	784	63
571	186	975	80	630	613
881	747	902	405	167	815
966	438	874	773	595	76
1764	1780	1924	1735	2084	1817

558	843	750	686	735	77
628	74	909	78	199	856
110	186	939	163	706	312
919	5	15	50	96	665
840	630	376	103	719	177
247	533	273	72	815	644
834	853	868	200	274	71
799	122	197	987	642	143
385	48	491	122	344	860
955	461	638	442	387	103
718	869	436	313	865	11
178	820	928	715	192	973
215	9	168	29	925	120
522	655	799	715	475	35
321	975	39	140	869	223
6	185	558	420	166	689
82	372	998	735	75	826
3	613	268	537	586	771
1664	1651	1930	1302	1814	1512

82	725	224	264	938	898
690	603	25	590	667	815
601	20	373	184	614	540
896	275	568	228	783	208
994	503	150	262	565	932
988	867	49	650	600	349
461	632	998	422	988	622
499	169	278	122	596	609
696	187	340	867	826	329
64	142	734	756	11	16
625	444	647	722	899	54
211	488	872	250	253	648
392	884	191	581	586	723
283	337	870	868	200	485
650	881	476	182	478	494
589	712	277	617	273	826
750	685	963	103	683	620
16	16	123	464	875	171
1898	1714	1632	1627	2167	1868

746	423	244	175	751	499
897	817	164	101	148	554
311	962	513	263	268	849
732	415	772	832	911	970
25	937	580	933	251	928
41	822	253	653	491	987
646	734	246	501	559	262
856	771	796	965	880	469
831	587	705	842	106	660
235	737	282	169	855	306
579	247	43	684	969	982
383	235	243	797	867	987
659	879	768	526	17	708
574	704	557	384	943	948
517	681	568	801	670	198
743	35	428	466	95	160
787	424	808	705	211	218
6	779	743	281	588	29
1914	2238	1743	2016	1916	2143

A.2 SAT instances

The following SAT instances were used in the experiments reported in Sections 4.4 and 6.2. These instances are in DIMACS CNF format. The first line is of the form "`p cnf` n m", where n is the number of variables in the instance (numbered 1 through n) and m is the number of clauses. Following this header line are m lines, one for each clause; each of these lines contains a set of positive or negative integers specifying the literals in the clause (for example, `3` denotes x_3 and `-3` denotes $\overline{x}_3$), followed by `0`.

A.2.1 Unsatisfiable bipartite matching: hall-set-15-10_01.cnf

This SAT instance represents an unsatisfiable bipartite matching instance. The underlying bipartite graph has 15 vertices on each side. Every vertex on the left-hand side has degree 9. The meaning of the variables in the SAT instance is as follows: For an edge joining vertex i on the left-hand side and a vertex j on the right-hand side, with $i, j \in \{0, 1, 2, \ldots, 14\}$, the variable $x_{15i+j+1}$ indicates whether the edge is part of the matching. (The header line specifies that the SAT instance contains 225 variables, but only 135 variables appear in the clauses; variables corresponding to non-edges in the underlying graph are irrelevant, do not appear in any clause, and therefore may take either value.) The first 15 clauses, which are of length 9, require that every vertex on the left-hand side is matched with at least one vertex on the right-hand side. The remaining 733 clauses, which are of length 2, require that no vertex on the right-hand side is matched with more than one vertex on the left-hand side. The instance is unsatisfiable because there is a subset of 10 vertices on the left-hand side that is adjacent to only 9 vertices on the right-hand side, thereby violating Hall's condition.

```
p cnf 225 748
2 3 4 5 8 11 12 13 15 0
16 18 19 22 23 25 27 28 30 0
32 33 34 35 38 41 42 43 45 0
46 47 49 52 53 56 57 58 60 0
62 63 64 65 68 71 72 73 75 0
77 78 79 80 83 86 87 88 90 0
92 93 94 95 98 101 102 103 105 0
107 108 109 110 113 116 117 118 120 0
123 125 126 128 129 130 131 133 134 0
136 137 139 141 144 146 147 148 149 0
```

```
151 152 153 155 159 160 161 164 165 0
167 168 169 170 173 176 177 178 180 0
182 183 184 185 188 191 192 193 195 0
197 198 199 200 203 206 207 208 210 0
212 213 214 215 218 221 222 223 225 0
-16 -46 0       -47 -152 0      -107 -212 0     -18 -153 0
-16 -136 0      -47 -167 0      -137 -152 0     -18 -168 0
-16 -151 0      -47 -182 0      -137 -167 0     -18 -183 0
-46 -136 0      -47 -197 0      -137 -182 0     -18 -198 0
-46 -151 0      -47 -212 0      -137 -197 0     -18 -213 0
-136 -151 0     -62 -77 0       -137 -212 0     -33 -63 0
-2 -32 0        -62 -92 0       -152 -167 0     -33 -78 0
-2 -47 0        -62 -107 0      -152 -182 0     -33 -93 0
-2 -62 0        -62 -137 0      -152 -197 0     -33 -108 0
-2 -77 0        -62 -152 0      -152 -212 0     -33 -123 0
-2 -92 0        -62 -167 0      -167 -182 0     -33 -153 0
-2 -107 0       -62 -182 0      -167 -197 0     -33 -168 0
-2 -137 0       -62 -197 0      -167 -212 0     -33 -183 0
-2 -152 0       -62 -212 0      -182 -197 0     -33 -198 0
-2 -167 0       -77 -92 0       -182 -212 0     -33 -213 0
-2 -182 0       -77 -107 0      -197 -212 0     -63 -78 0
-2 -197 0       -77 -137 0      -3 -18 0        -63 -93 0
-2 -212 0       -77 -152 0      -3 -33 0        -63 -108 0
-32 -47 0       -77 -167 0      -3 -63 0        -63 -123 0
-32 -62 0       -77 -182 0      -3 -78 0        -63 -153 0
-32 -77 0       -77 -197 0      -3 -93 0        -63 -168 0
-32 -92 0       -77 -212 0      -3 -108 0       -63 -183 0
-32 -107 0      -92 -107 0      -3 -123 0       -63 -198 0
-32 -137 0      -92 -137 0      -3 -153 0       -63 -213 0
-32 -152 0      -92 -152 0      -3 -168 0       -78 -93 0
-32 -167 0      -92 -167 0      -3 -183 0       -78 -108 0
-32 -182 0      -92 -182 0      -3 -198 0       -78 -123 0
-32 -197 0      -92 -197 0      -3 -213 0       -78 -153 0
-32 -212 0      -92 -212 0      -18 -33 0       -78 -168 0
-47 -62 0       -107 -137 0     -18 -63 0       -78 -183 0
-47 -77 0       -107 -152 0     -18 -78 0       -78 -198 0
-47 -92 0       -107 -167 0     -18 -93 0       -78 -213 0
-47 -107 0      -107 -182 0     -18 -108 0      -93 -108 0
-47 -137 0      -107 -197 0     -18 -123 0      -93 -123 0
```

```
-93 -153 0      -19 -64 0       -79 -184 0      -35 -155 0
-93 -168 0      -19 -79 0       -79 -199 0      -35 -170 0
-93 -183 0      -19 -94 0       -79 -214 0      -35 -185 0
-93 -198 0      -19 -109 0      -94 -109 0      -35 -200 0
-93 -213 0      -19 -139 0      -94 -139 0      -35 -215 0
-108 -123 0     -19 -169 0      -94 -169 0      -65 -80 0
-108 -153 0     -19 -184 0      -94 -184 0      -65 -95 0
-108 -168 0     -19 -199 0      -94 -199 0      -65 -110 0
-108 -183 0     -19 -214 0      -94 -214 0      -65 -125 0
-108 -198 0     -34 -49 0       -109 -139 0     -65 -155 0
-108 -213 0     -34 -64 0       -109 -169 0     -65 -170 0
-123 -153 0     -34 -79 0       -109 -184 0     -65 -185 0
-123 -168 0     -34 -94 0       -109 -199 0     -65 -200 0
-123 -183 0     -34 -109 0      -109 -214 0     -65 -215 0
-123 -198 0     -34 -139 0      -139 -169 0     -80 -95 0
-123 -213 0     -34 -169 0      -139 -184 0     -80 -110 0
-153 -168 0     -34 -184 0      -139 -199 0     -80 -125 0
-153 -183 0     -34 -199 0      -139 -214 0     -80 -155 0
-153 -198 0     -34 -214 0      -169 -184 0     -80 -170 0
-153 -213 0     -49 -64 0       -169 -199 0     -80 -185 0
-168 -183 0     -49 -79 0       -169 -214 0     -80 -200 0
-168 -198 0     -49 -94 0       -184 -199 0     -80 -215 0
-168 -213 0     -49 -109 0      -184 -214 0     -95 -110 0
-183 -198 0     -49 -139 0      -199 -214 0     -95 -125 0
-183 -213 0     -49 -169 0      -5 -35 0        -95 -155 0
-198 -213 0     -49 -184 0      -5 -65 0        -95 -170 0
-4 -19 0        -49 -199 0      -5 -80 0        -95 -185 0
-4 -34 0        -49 -214 0      -5 -95 0        -95 -200 0
-4 -49 0        -64 -79 0       -5 -110 0       -95 -215 0
-4 -64 0        -64 -94 0       -5 -125 0       -110 -125 0
-4 -79 0        -64 -109 0      -5 -155 0       -110 -155 0
-4 -94 0        -64 -139 0      -5 -170 0       -110 -170 0
-4 -109 0       -64 -169 0      -5 -185 0       -110 -185 0
-4 -139 0       -64 -184 0      -5 -200 0       -110 -200 0
-4 -169 0       -64 -199 0      -5 -215 0       -110 -215 0
-4 -184 0       -64 -214 0      -35 -65 0       -125 -155 0
-4 -199 0       -79 -94 0       -35 -80 0       -125 -170 0
-4 -214 0       -79 -109 0      -35 -95 0       -125 -185 0
-19 -34 0       -79 -139 0      -35 -110 0      -125 -200 0
-19 -49 0       -79 -169 0      -35 -125 0      -125 -215 0
```

```
-155 -170 0
-155 -185 0
-155 -200 0
-155 -215 0
-170 -185 0
-170 -200 0
-170 -215 0
-185 -200 0
-185 -215 0
-200 -215 0
-126 -141 0
-22 -52 0
-8 -23 0
-8 -38 0
-8 -53 0
-8 -68 0
-8 -83 0
-8 -98 0
-8 -113 0
-8 -128 0
-8 -173 0
-8 -188 0
-8 -203 0
-8 -218 0
-23 -38 0
-23 -53 0
-23 -68 0
-23 -83 0
-23 -98 0
-23 -113 0
-23 -128 0
-23 -173 0
-23 -188 0
-23 -203 0
-23 -218 0
-38 -53 0
-38 -68 0
-38 -83 0
-38 -98 0
-38 -113 0
-38 -128 0
-38 -173 0
-38 -188 0
-38 -203 0
-38 -218 0
-53 -68 0
-53 -83 0
-53 -98 0
-53 -113 0
-53 -128 0
-53 -173 0
-53 -188 0
-53 -203 0
-53 -218 0
-68 -83 0
-68 -98 0
-68 -113 0
-68 -128 0
-68 -173 0
-68 -188 0
-68 -203 0
-68 -218 0
-83 -98 0
-83 -113 0
-83 -128 0
-83 -173 0
-83 -188 0
-83 -203 0
-83 -218 0
-98 -113 0
-98 -128 0
-98 -173 0
-98 -188 0
-98 -203 0
-98 -218 0
-113 -128 0
-113 -173 0
-113 -188 0
-113 -203 0
-113 -218 0
-128 -173 0
-128 -188 0
-128 -203 0
-128 -218 0
-173 -188 0
-173 -203 0
-173 -218 0
-188 -203 0
-188 -218 0
-203 -218 0
-129 -144 0
-129 -159 0
-144 -159 0
-25 -130 0
-25 -160 0
-130 -160 0
-11 -41 0
-11 -56 0
-11 -71 0
-11 -86 0
-11 -101 0
-11 -116 0
-11 -131 0
-11 -146 0
-11 -161 0
-11 -176 0
-11 -191 0
-11 -206 0
-11 -221 0
-41 -56 0
-41 -71 0
-41 -86 0
-41 -101 0
-41 -116 0
-41 -131 0
-41 -146 0
-41 -161 0
-41 -176 0
-41 -191 0
-41 -206 0
-41 -221 0
-56 -71 0
-56 -86 0
-56 -101 0
-56 -116 0
-56 -131 0
-56 -146 0
-56 -161 0
-56 -176 0
-56 -191 0
-56 -206 0
-56 -221 0
-71 -86 0
-71 -101 0
-71 -116 0
-71 -131 0
-71 -146 0
-71 -161 0
-71 -176 0
-71 -191 0
-71 -206 0
-71 -221 0
-86 -101 0
-86 -116 0
-86 -131 0
-86 -146 0
-86 -161 0
-86 -176 0
-86 -191 0
-86 -206 0
-86 -221 0
-101 -116 0
-101 -131 0
-101 -146 0
-101 -161 0
-101 -176 0
-101 -191 0
-101 -206 0
-101 -221 0
-116 -131 0
```

```
-116 -146 0     -27 -57 0       -87 -177 0      -28 -73 0
-116 -161 0     -27 -72 0       -87 -192 0      -28 -88 0
-116 -176 0     -27 -87 0       -87 -207 0      -28 -103 0
-116 -191 0     -27 -102 0      -87 -222 0      -28 -118 0
-116 -206 0     -27 -117 0      -102 -117 0     -28 -133 0
-116 -221 0     -27 -147 0      -102 -147 0     -28 -148 0
-131 -146 0     -27 -177 0      -102 -177 0     -28 -178 0
-131 -161 0     -27 -192 0      -102 -192 0     -28 -193 0
-131 -176 0     -27 -207 0      -102 -207 0     -28 -208 0
-131 -191 0     -27 -222 0      -102 -222 0     -28 -223 0
-131 -206 0     -42 -57 0       -117 -147 0     -43 -58 0
-131 -221 0     -42 -72 0       -117 -177 0     -43 -73 0
-146 -161 0     -42 -87 0       -117 -192 0     -43 -88 0
-146 -176 0     -42 -102 0      -117 -207 0     -43 -103 0
-146 -191 0     -42 -117 0      -117 -222 0     -43 -118 0
-146 -206 0     -42 -147 0      -147 -177 0     -43 -133 0
-146 -221 0     -42 -177 0      -147 -192 0     -43 -148 0
-161 -176 0     -42 -192 0      -147 -207 0     -43 -178 0
-161 -191 0     -42 -207 0      -147 -222 0     -43 -193 0
-161 -206 0     -42 -222 0      -177 -192 0     -43 -208 0
-161 -221 0     -57 -72 0       -177 -207 0     -43 -223 0
-176 -191 0     -57 -87 0       -177 -222 0     -58 -73 0
-176 -206 0     -57 -102 0      -192 -207 0     -58 -88 0
-176 -221 0     -57 -117 0      -192 -222 0     -58 -103 0
-191 -206 0     -57 -147 0      -207 -222 0     -58 -118 0
-191 -221 0     -57 -177 0      -13 -28 0       -58 -133 0
-206 -221 0     -57 -192 0      -13 -43 0       -58 -148 0
-12 -27 0       -57 -207 0      -13 -58 0       -58 -178 0
-12 -42 0       -57 -222 0      -13 -73 0       -58 -193 0
-12 -57 0       -72 -87 0       -13 -88 0       -58 -208 0
-12 -72 0       -72 -102 0      -13 -103 0      -58 -223 0
-12 -87 0       -72 -117 0      -13 -118 0      -73 -88 0
-12 -102 0      -72 -147 0      -13 -133 0      -73 -103 0
-12 -117 0      -72 -177 0      -13 -148 0      -73 -118 0
-12 -147 0      -72 -192 0      -13 -178 0      -73 -133 0
-12 -177 0      -72 -207 0      -13 -193 0      -73 -148 0
-12 -192 0      -72 -222 0      -13 -208 0      -73 -178 0
-12 -207 0      -87 -102 0      -13 -223 0      -73 -193 0
-12 -222 0      -87 -117 0      -28 -43 0       -73 -208 0
-27 -42 0       -87 -147 0      -28 -58 0       -73 -223 0
```

```
-88 -103 0      -178 -193 0     -30 -210 0      -90 -120 0
-88 -118 0      -178 -208 0     -30 -225 0      -90 -165 0
-88 -133 0      -178 -223 0     -45 -60 0       -90 -180 0
-88 -148 0      -193 -208 0     -45 -75 0       -90 -195 0
-88 -178 0      -193 -223 0     -45 -90 0       -90 -210 0
-88 -193 0      -208 -223 0     -45 -105 0      -90 -225 0
-88 -208 0      -134 -149 0     -45 -120 0      -105 -120 0
-88 -223 0      -134 -164 0     -45 -165 0      -105 -165 0
-103 -118 0     -149 -164 0     -45 -180 0      -105 -180 0
-103 -133 0     -15 -30 0       -45 -195 0      -105 -195 0
-103 -148 0     -15 -45 0       -45 -210 0      -105 -210 0
-103 -178 0     -15 -60 0       -45 -225 0      -105 -225 0
-103 -193 0     -15 -75 0       -60 -75 0       -120 -165 0
-103 -208 0     -15 -90 0       -60 -90 0       -120 -180 0
-103 -223 0     -15 -105 0      -60 -105 0      -120 -195 0
-118 -133 0     -15 -120 0      -60 -120 0      -120 -210 0
-118 -148 0     -15 -165 0      -60 -165 0      -120 -225 0
-118 -178 0     -15 -180 0      -60 -180 0      -165 -180 0
-118 -193 0     -15 -195 0      -60 -195 0      -165 -195 0
-118 -208 0     -15 -210 0      -60 -210 0      -165 -210 0
-118 -223 0     -15 -225 0      -60 -225 0      -165 -225 0
-133 -148 0     -30 -45 0       -75 -90 0       -180 -195 0
-133 -178 0     -30 -60 0       -75 -105 0      -180 -210 0
-133 -193 0     -30 -75 0       -75 -120 0      -180 -225 0
-133 -208 0     -30 -90 0       -75 -165 0      -195 -210 0
-133 -223 0     -30 -105 0      -75 -180 0      -195 -225 0
-148 -178 0     -30 -120 0      -75 -195 0      -210 -225 0
-148 -193 0     -30 -165 0      -75 -210 0
-148 -208 0     -30 -180 0      -75 -225 0
-148 -223 0     -30 -195 0      -90 -105 0
```

A.2.2 Unsatisfiable biconditional formula: gz4.cnf

Groote and Zantema [44] describe a particular iterative procedure for generating unsatisfiable biconditional formulas. This SAT instance was produced from the formula $\neg[S_4]$ generated on the fourth iteration of this construction; it was converted to conjunctive normal form by applying the Tseitin transformation (see Section 6.1.1).

```
p cnf 95 253
1 2 3 0
-1 -2 3 0
-1 2 -3 0
1 -2 -3 0
3 4 5 0
-3 -4 5 0
-3 4 -5 0
3 -4 -5 0
5 6 7 0
-5 -6 7 0
-5 6 -7 0
5 -6 -7 0
7 8 9 0
-7 -8 9 0
-7 8 -9 0
7 -8 -9 0
9 10 11 0
-9 -10 11 0
-9 10 -11 0
9 -10 -11 0
11 12 13 0
-11 -12 13 0
-11 12 -13 0
11 -12 -13 0
6 13 14 0
-6 -13 14 0
6 -13 -14 0
-6 13 -14 0
14 15 16 0
-14 -15 16 0
-14 15 -16 0
14 -15 -16 0
16 17 18 0
-16 -17 18 0
-16 17 -18 0
16 -17 -18 0
4 18 19 0
-4 -18 19 0
4 -18 -19 0
-4 18 -19 0
19 20 21 0
-19 -20 21 0
-19 20 -21 0
19 -20 -21 0
21 22 23 0
-21 -22 23 0
-21 22 -23 0
21 -22 -23 0
23 24 25 0
-23 -24 25 0
-23 24 -25 0
23 -24 -25 0
12 25 26 0
-12 -25 26 0
12 -25 -26 0
-12 25 -26 0
20 26 27 0
-20 -26 27 0
20 -26 -27 0
-20 26 -27 0
27 28 29 0
-27 -28 29 0
-27 28 -29 0
27 -28 -29 0
2 29 30 0
-2 -29 30 0
2 -29 -30 0
-2 29 -30 0
30 31 32 0
-30 -31 32 0
-30 31 -32 0
30 -31 -32 0
32 33 34 0
-32 -33 34 0
-32 33 -34 0
32 -33 -34 0
34 35 36 0
-34 -35 36 0
-34 35 -36 0
34 -35 -36 0
10 36 37 0
-10 -36 37 0
10 -36 -37 0
-10 36 -37 0
37 38 39 0
-37 -38 39 0
-37 38 -39 0
37 -38 -39 0
33 39 40 0
-33 -39 40 0
33 -39 -40 0
-33 39 -40 0
40 41 42 0
-40 -41 42 0
-40 41 -42 0
40 -41 -42 0
17 42 43 0
-17 -42 43 0
17 -42 -43 0
-17 42 -43 0
31 43 44 0
-31 -43 44 0
31 -43 -44 0
-31 43 -44 0
44 45 46 0
-44 -45 46 0
-44 45 -46 0
44 -45 -46 0
46 47 48 0
-46 -47 48 0
-46 47 -48 0
46 -47 -48 0
24 48 49 0
-24 -48 49 0
24 -48 -49 0
-24 48 -49 0
38 49 50 0
-38 -49 50 0
38 -49 -50 0
-38 49 -50 0
45 50 51 0
-45 -50 51 0
45 -50 -51 0
-45 50 -51 0
1 51 52 0
-1 -51 52 0
1 -51 -52 0
-1 51 -52 0
52 53 54 0
-52 -53 54 0
-52 53 -54 0
52 -53 -54 0
54 55 56 0
-54 -55 56 0
-54 55 -56 0
54 -55 -56 0
56 57 58 0
-56 -57 58 0
-56 57 -58 0
56 -57 -58 0
8 58 59 0
-8 -58 59 0
8 -58 -59 0
-8 58 -59 0
59 60 61 0
-59 -60 61 0
-59 60 -61 0
59 -60 -61 0
61 62 63 0
-61 -62 63 0
-61 62 -63 0
61 -62 -63 0
57 63 64 0
-57 -63 64 0
57 -63 -64 0
-57 63 -64 0
15 64 65 0
-15 -64 65 0
15 -64 -65 0
```

```
-15 64 -65 0    -62 73 -74 0    -35 81 -82 0    -78 88 -89 0
65 66 67 0      69 74 75 0      60 82 83 0      89 90 91 0
-65 -66 67 0    -69 -74 75 0    -60 -82 83 0    -89 -90 91 0
-65 66 -67 0    69 -74 -75 0    60 -82 -83 0    -89 90 -91 0
65 -66 -67 0    -69 74 -75 0    -60 82 -83 0    89 -90 -91 0
55 67 68 0      28 75 76 0      83 84 85 0      47 91 92 0
-55 -67 68 0    -28 -75 76 0    -83 -84 85 0    -47 -91 92 0
55 -67 -68 0    28 -75 -76 0    -83 84 -85 0    47 -91 -92 0
-55 67 -68 0    -28 75 -76 0    83 -84 -85 0    -47 91 -92 0
68 69 70 0      53 76 77 0      80 85 86 0      72 92 93 0
-68 -69 70 0    -53 -76 77 0    -80 -85 86 0    -72 -92 93 0
-68 69 -70 0    53 -76 -77 0    80 -85 -86 0    72 -92 -93 0
68 -69 -70 0    -53 76 -77 0    -80 85 -86 0    -72 92 -93 0
22 70 71 0      77 78 79 0      41 86 87 0      84 93 94 0
-22 -70 71 0    -77 -78 79 0    -41 -86 87 0    -84 -93 94 0
22 -70 -71 0    -77 78 -79 0    41 -86 -87 0    84 -93 -94 0
-22 70 -71 0    77 -78 -79 0    -41 86 -87 0    -84 93 -94 0
71 72 73 0      79 80 81 0      66 87 88 0      90 94 95 0
-71 -72 73 0    -79 -80 81 0    -66 -87 88 0    -90 -94 95 0
-71 72 -73 0    -79 80 -81 0    66 -87 -88 0    90 -94 -95 0
71 -72 -73 0    79 -80 -81 0    -66 87 -88 0    -90 94 -95 0
62 73 74 0      35 81 82 0      78 88 89 0      -95 0
-62 -73 74 0    -35 -81 82 0    -78 -88 89 0
62 -73 -74 0    35 -81 -82 0    78 -88 -89 0
```

A.2.3 Unsatisfiable bipartite matching: hall-set-10-4_1.cnf

This SAT instance, whose constraint graph appears in Section 6.2, represents an unsatisfiable bipartite matching instance. The construction is identical to that of hall-set-15-10_01.cnf above, except that the underlying graph is smaller: there are 10 vertices on each side, and each vertex on the left-hand side has degree 4. The first five vertices on the left-hand side are adjacent only to the first four vertices on the right-hand side, thus forming a subset that violates Hall's condition.

```
p cnf 100 104   41 42 43 44 0   92 94 95 97 0   -1 -61 0
1 2 3 4 0       53 55 56 60 0   -1 -11 0        -11 -21 0
11 12 13 14 0   61 64 67 70 0   -1 -21 0        -11 -31 0
21 22 23 24 0   73 78 79 80 0   -1 -31 0        -11 -41 0
31 32 33 34 0   83 84 85 86 0   -1 -41 0        -11 -61 0
```

```
-21 -31 0       -3 -23 0        -43 -73 0       -24 -94 0
-21 -41 0       -3 -33 0        -43 -83 0       -34 -44 0
-21 -61 0       -3 -43 0        -53 -73 0       -34 -64 0
-31 -41 0       -3 -53 0        -53 -83 0       -34 -84 0
-31 -61 0       -3 -73 0        -73 -83 0       -34 -94 0
-41 -61 0       -3 -83 0        -4 -14 0        -44 -64 0
-2 -12 0        -13 -23 0       -4 -24 0        -44 -84 0
-2 -22 0        -13 -33 0       -4 -34 0        -44 -94 0
-2 -32 0        -13 -43 0       -4 -44 0        -64 -84 0
-2 -42 0        -13 -53 0       -4 -64 0        -64 -94 0
-2 -92 0        -13 -73 0       -4 -84 0        -84 -94 0
-12 -22 0       -13 -83 0       -4 -94 0        -55 -85 0
-12 -32 0       -23 -33 0       -14 -24 0       -55 -95 0
-12 -42 0       -23 -43 0       -14 -34 0       -85 -95 0
-12 -92 0       -23 -53 0       -14 -44 0       -56 -86 0
-22 -32 0       -23 -73 0       -14 -64 0       -67 -97 0
-22 -42 0       -23 -83 0       -14 -84 0       -60 -70 0
-22 -92 0       -33 -43 0       -14 -94 0       -60 -80 0
-32 -42 0       -33 -53 0       -24 -34 0       -70 -80 0
-32 -92 0       -33 -73 0       -24 -44 0
-42 -92 0       -33 -83 0       -24 -64 0
-3 -13 0        -43 -53 0       -24 -84 0
```

Bibliography

[1] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, March 2007.

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

[3] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 448–456, Montreal, May 2002.

[4] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP '07)*, pages 118–132, Providence, September 2007.

[5] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011.

[6] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS '06)*, pages 697–708, Berkeley, October 2006.

[7] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. A new approximation method for set covering problems, with applications to multidimensional bin packing. *SIAM Journal on Computing*, 39(4): 1256–1278, 2010.

[8] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, December 2004.

[9] James E. Beck and Daniel P. Siewiorek. Modeling multicomputer task allocation as a vector packing problem. In *Proceedings of the 9th International Symposium on System Synthesis (ISSS '96)*, pages 115–120, La Jolla, November 1996.

[10] Markus Behle. On threshold BDDs and the optimal variable ordering problem. *Journal of Combinatorial Optimization*, 16(2):107–118, August 2008.

[11] David Bergman, Willem-Jan van Hoeve, and J. N. Hooker. Manipulating MDD relaxations for combinatorial optimization. In *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011)*, pages 20–35, Berlin, May 2011.

[12] David Bergman, Andre A. Cire, and Willem-Jan van Hoeve. MDD propagation for sequence constraints. *Journal of Artificial Intelligence Research*, 50:697–722, July 2014.

[13] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and J. N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2):253–268, Spring 2014.

[14] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and Tallys Yunes. BDD-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, April 2014.

[15] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Design Automation Conference (DAC '99)*, pages 317–320, Atlanta, October 1999.

[16] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, pages 454–464, Paris, July 2001.

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

[18] Maria Luisa Bonet and Nicola Galesi. Optimality of size-width tradeoffs for resolution. *Computational Complexity*, 10(4):261–276, December 2001.

[19] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM Journal on Computing*, 30(5): 1462–1484, 2000.

[20] George Boole. *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly, London, 1854.

[21] Daniel Brand. Verification of large synthesized designs. In *Digest of Technical Papers of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '93)*, pages 534–537, Santa Clara, November 1993.

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

[23] Joshua Buresh-Oppenheim and Toniann Pitassi. The complexity of resolution refinements. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 138–147, Ottawa, June 2003.

[24] Alberto Caprara and Paolo Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, August 2001.

[25] Soo Y. Chang, Hark-Chin Hwang, and Sanghyuck Park. A two-dimensional vector packing model for the efficient use of coil cassettes. *Computers & Operations Research*, 32(8):2051–2058, August 2005.

[26] Pinhong Chen and Kurt Keutzer. Towards true crosstalk noise analysis. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '99)*, pages 132–138, San Jose, November 1999.

[27] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on*

Mathematical Software, 35(3):22:1–14, October 2008. Available as part of the SuiteSparse library, `http://faculty.cse.tamu.edu/davis/suitesparse.html`.

[28] Andre A. Cire and Willem-Jan van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, December 2013.

[29] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, pages 168–176, Barcelona, March–April 2004.

[30] Edward G. Coffman, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Survey and classification. In Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 455–531. Springer, second edition, 2013.

[31] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, Shaker Heights, May 1971.

[32] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining flow-based propagation. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012)*, pages 146–162, Nantes, May–June 2012.

[33] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, Washington, 1984.

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

[35] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, pages 61–75, St. Andrews, June 2005.

[36] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Selected Revised Papers of the 6th International Conference on Theory and Ap-*

plications of Satisfiability Testing (SAT 2003), pages 502–518, Santa Margherita Ligure, May 2004.

[37] Samuel Eilon and Nicos Christofides. The loading problem. *Management Science*, 17(5):259–268, January 1971.

[38] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, pages 340–354, Lisbon, May 2007.

[39] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[40] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing (STOC '72)*, pages 143–150, Denver, May 1972.

[41] M. R. Garey, R. L. Graham, D. S. Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A*, 21(3):257–298, November 1976.

[42] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS '98)*, pages 208–213, Pittsburgh, June 1998.

[43] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 20, pages 633–654. IOS Press, 2009.

[44] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, August 2003.

[45] Tarik Hadzic, John N. Hooker, Barry. O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Proceedings of the 14th International Conference on*

Principles and Practice of Constraint Programming (CP 2008), pages 448–462, Sydney, September 2008.

[46] Paul Heckbert. Color image quantization for frame buffer display. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '82)*, pages 297–307, Boston, July 1982.

[47] Samid Hoda, Willem-Jan van Hoeve, and J. N. Hooker. A systematic approach to MDD-based constraint programming. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP 2010)*, pages 266–280, St. Andrews, September 2010.

[48] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In *Proceedings of the 3rd Workshop on Satisfiability (SAT 2000)*, pages 283–292, Renesse, May 2000. SATLIB is available online at `http://www.satlib.org/`.

[49] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, Cambridge, 1985.

[50] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 730–733, Limerick, June 2000.

[51] David S. Johnson. Fast allocation algorithms. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT '72)*, pages 144–154, October 1972.

[52] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, December 1974.

[53] George Katsirelos. *Nogood Processing in CSPs*. Ph.D. thesis, Graduate Department of Computer Science, University of Toronto, 2008.

[54] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI-13)*, pages 481–488, Bellevue, July 2013.

[55] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, Vienna, August 1992.

[56] Brian Kell and Willem-Jan van Hoeve. An MDD approach to multidimensional bin packing. In *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013)*, pages 128–143, Yorktown Heights, May 2013.

[57] Brian Kell, Ashish Sabharwal, and Willem-Jan van Hoeve. BDD-guided clause generation. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*, pages 215–230, Barcelona, May 2015.

[58] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, October 2004.

[59] Fatma Kılınç Karzan, George L. Nemhauser, and Martin W. P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, December 2009.

[60] Donald E. Knuth. *The Art of Computer Programming*, volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley, Upper Saddle River, 2011.

[61] Haluk Konuk and Tracy Larrabee. Explorations of sequential ATPG using Boolean satisfiability. In *Digest of Papers of the 11th Annual IEEE VLSI Test Symposium (VTS '93)*, pages 85–90, Atlantic City, April 1993.

[62] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, December 2002.

[63] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, January 1992.

[64] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, July 1959.

[65] R. Lewis, X. Song, K. Dowsland, and J. Thompson. An investigation into two bin packing problems with ordering and orientation implications. *European Journal of Operational Research*, 213(1):52–65, August 2011.

[66] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, September 2002.

[67] Inês Lynce and João Marques-Silva. Efficient haplotype inference with Boolean satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, page 104, Boston, July 2006.

[68] João Marques-Silva. Practical applications of Boolean satisfiability. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES 2008)*, pages 74–80, Göteborg, May 2008.

[69] João Marques-Silva and Thomas Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the 1999 Design, Automation and Test in Europe Conference and Exhibition (DATE '99)*, pages 145–149, Munich, March 1999.

[70] João Marques-Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *Digest of Papers of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 152–161, Seattle, June 1997.

[71] João Marques-Silva, Inês Lynce, and Sharad Malik. CDCL solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 4, pages 131–154. IOS Press, 2009.

[72] João P. Marques-Silva and Karem A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '96)*, pages 220–227, San Jose, November 1996.

[73] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[74] Silvano Martello, David Pisinger, and Daniele Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, April 2000.

[75] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, pages 1–13, Boulder, July 2003.

[76] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD—Foundations and Applications*. Springer, Berlin, 1998.

[77] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference (DAC '93)*, pages 272–277, Dallas, June 1993.

[78] Shin-ichi Minato. Calculation of unate cube set algebra using zero-suppressed BDDs. In *Proceedings of the 31st Design Automation Conference (DAC '94)*, pages 420–424, San Diego, June 1994.

[79] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, Las Vegas, June 2001.

[80] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009.

[81] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175 (2):512–525, February 2011.

[82] Jean-Charles Régin and Mohamed Rezgui. Discussion about constraint programming bin packing models. In *AI for Data Center Management and Cloud Computing: Papers from the Workshop at the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, pages 21–23, San Francisco, August 2011.

[83] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, September 2006.

[84] Pierre Schaus. *Solving Balancing and Bin-Packing Problems with Constraint Programming*. Doctoral thesis, Ecole polytechnique de Louvain, Département d'Ingénierie Informatique, Université catholique de Louvain, Louvain-la-Neuve, August 2009.

[85] Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving Steel Mill Slab Problems with constraint-based techniques: CP, LNS, and CBLS. *Constraints*, 16(2):125–147, April 2011.

[86] Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. Cardinality reasoning for bin-packing constraint: Application to a tank allocation problem. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, pages 815–822, Québec City, October 2012.

[87] Hadas Shachnai and Tami Tamir. Approximation schemes for generalized two-dimensional vector packing with application to data placement. *Journal of Discrete Algorithms*, 10:35–48, January 2012.

[88] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12): 713–723, 1938.

[89] Paul Shaw. A constraint for bin packing. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 648–662, Toronto, September–October 2004.

[90] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, pages 127–144, Austin, November 2000.

[91] Frits C. R. Spieksma. A branch-and-bound algorithm for the two-dimensional vector packing problem. *Computers & Operations Research*, 21(1):19–25, January 1994.

[92] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Digest of Technical Papers of the 1990 IEEE International Conference on Computer-Aided Design (ICCAD-90)*, pages 92–95, Santa Clara, November 1990.

[93] Paul Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, September 1996.

[94] Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, pages 5–9, Bologna, June 2010.

[95] G. S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, volume 2: Classical Papers on Computational Logic 1967–1970, pages 466–483. Springer, 1983.

[96] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.

[97] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, December 2007.

[98] Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[99] Hantao Zhang and Jieh Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of the 1994 International Computer Symposium (ICS '94)*, Hsinchu, December 1994.