Knowledge Compilation for Solving Computationally Hard Problems

Permalink
https://escholarship.org/uc/item/2vz8n06q

Author
Oztok, Umut

Publication Date
2017

Peer reviewed|Thesis/dissertation
Knowledge Compilation for Solving Computationally Hard Problems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Umut Oztok

2017
ABSTRACT OF THE DISSERTATION

Knowledge Compilation for Solving Computationally Hard Problems

by

Umut Oztok
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017
Professor Adnan Youssef Darwiche, Chair

Knowledge compilation is concerned with compiling problems encoded in some input language into some tractable, output language, with the goal of allowing one to solve such problems efficiently if the compilation is successful. This paradigm was originally motivated by the need to push much of the computational overhead into an offline compilation phase, which can then be amortized over a large number of queries in an online computation phase. In this dissertation, we study various new approaches to enhance the offline compilation phase, both theoretically and practically. We also study knowledge compilation from a perspective where it is employed as a general methodology for computation instead of just a paradigm that is concerned with the offline/online divide.

In particular, we introduce a hierarchy of complexity parameters to bound the sizes of compiled representations. These new parameters are based on incorporating the logical content of the input representations, as opposed to existing parameters (e.g., treewidth) that are only based on the structure of the input. Our results improve some of the best known upper bounds on the compilation of influential representations, such as DNNFs, SDDs, and OBDDs. Moreover, we develop two new practical compilation algorithms for the DNNF and SDD languages, leading to orders of magnitude faster compilations. Finally, we study solving Beyond-NP problems using knowledge compilation, while particularly ex-
tending the reach of knowledge compilation to tackling problems in the highly intractable complexity class $\text{PP}^\text{PP}$. Our results show the applicability of knowledge compilers as black-box tools for solving Beyond-$\text{NP}$ problems, similar to the use of SAT solvers for solving $\text{NP}$-complete problems.
The dissertation of Umut Oztok is approved.

Richard E. Korf

Todd D. Millstein

Paulo Tabuada

Adnan Youssef Darwiche, Committee Chair

University of California, Los Angeles

2017
# Table of Contents

1. **Introduction** ......................................................... 1

2. **Technical Preliminaries** ........................................... 8
   2.1 Propositional Logic ................................................... 8
      2.1.1 Syntax .......................................................... 8
      2.1.2 Semantics ...................................................... 9
   2.2 Boolean Functions ................................................. 12
      2.2.1 Conjunctive Normal Form .................................... 13
   2.3 Knowledge Compilation ........................................... 14
      2.3.1 Target Languages ............................................. 15
      2.3.2 Tractability of Representations ............................. 18
      2.3.3 Succinctness of Representations ............................. 19
      2.3.4 Sentential Decision Diagram (SDD) ........................ 21
   2.4 Beyond \( \text{NP} \) .................................................. 24

3. **CV-Width: A New Complexity Parameter for CNFs** ............ 29
   3.1 Introduction ....................................................... 29
   3.2 Structured DNNF ...................................................) 31
   3.3 Decomposing CNFs ................................................ 31
   3.4 Constructing Decompositions .................................... 33
      3.4.1 Decomposition by Splitting on Variables .................. 33
      3.4.2 Decomposition by Splitting on Clauses .................... 34
   3.5 More on Vtrees .................................................... 35
3.6 Compiling CNFs into Structured DNNF ........................................... 35
3.7 A New Complexity Parameter for CNFs ............................................. 38
  3.7.1 Counting Components ......................................................... 38
  3.7.2 Clause-Variable Width ....................................................... 40
  3.7.3 Complexity Analysis .......................................................... 42

Appendices ................................................................. 45
  3.A Proof of Lemma 1 .............................................................. 45

4 On Compiling CNF into Deterministic Subsets of DNNF ... 48
  4.1 Introduction ................................................................. 48
  4.2 Compiling CNFs into Decision-DNNFs ....................................... 51
    4.2.1 Decision-DNNF .......................................................... 51
    4.2.2 Decision Vtrees ......................................................... 52
    4.2.3 A Compilation Algorithm .......................................... 53
    4.2.4 Decision-Width ......................................................... 56
  4.3 Decision-DNNFs and Model Counters ....................................... 58
  4.4 From Decision-DNNF to SDD ............................................. 60
  4.5 Compiling CNFs into OBDDs .............................................. 63

5 A Comparison of New and Classical CNF Parameters ...... 64
  5.1 Classical CNF Parameters .................................................. 65
    5.1.1 Treewidth ............................................................... 65
    5.1.2 Cutwidth and Pathwidth ........................................... 69
  5.2 A Comparison of New and Classical CNF Parameters .................. 70
    5.2.1 CV-Width and Incidence Treewidth .............................. 70
5.2.2 Decision-Width and Primal Treewidth .................. 71
5.2.3 Linear CV-Width, Cutwidth, and Pathwidth ............ 72
5.3 A Discussion on the Comparison ................................. 74

Appendices ........................................................................ 78
5.A Proof of Theorem 5 ....................................................... 78
5.B Proof of Theorem 7 ....................................................... 88

6 A Top-Down Compiler for Sentential Decision Diagrams .... 90
6.1 Introduction ................................................................. 90
6.2 Decision-SDDs ............................................................. 92
6.3 A Formal Framework for the Compiler ......................... 95
6.4 A Top-Down SDD Compiler .......................................... 98
6.5 A Top-Down Model Counter ........................................ 104
6.5.1 Weighted Model Counting ....................................... 104
6.6 Experiments ................................................................. 105
6.6.1 Top-Down Compiler ................................................ 105
6.6.2 Top-Down Model Counter ....................................... 108
6.7 Related Work .............................................................. 109

Appendices ........................................................................ 112
6.A Quasipolynomial Simulation of Decision-SDDs ................ 112
6.B Soundness of the Compilation Algorithm ....................... 115
6.C Computing Cache Key ............................................... 126

7 On Compiling DNNFs without Determinism .................... 127
7.1 Introduction .......................................................... 127
7.2 Compiling DNNFs through Forgetting Variables ................. 130
7.3 An Exponential Separation by Forgetting Variables .............. 131
7.4 EMF Transformations .................................................. 134
   7.4.1 Tseitin Transformation ...................................... 134
   7.4.2 Extended Resolution .......................................... 136
   7.4.3 Bounded Variable Addition .................................. 138
7.5 Experiments .................................................................. 141
7.6 Related Work ................................................................ 142

Appendices ...................................................................... 145

7.A Proof of Theorem 24 ..................................................... 145
7.B Proof of Theorem 25 ..................................................... 146

8 Solving PP_{p}^{p}-Complete Problems Using Knowledge Compilation147
  8.1 Introduction ............................................................. 147
  8.2 Technical Background ............................................... 149
  8.3 Solving MAJMAJSAT using SDDs ................................. 150
     8.3.1 Problem Definition ............................................ 151
     8.3.2 Constrained SDDs .............................................. 151
     8.3.3 A New Algorithm to Solve MAJMAJSAT .................. 152
  8.4 Computing the SDP using SDDs .................................... 155
     8.4.1 SDP on Weighted Boolean Functions ....................... 155
     8.4.2 A New Algorithm to Compute SDPs ....................... 157
  8.5 Compiling X-Constrained SDDs .................................... 160
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A pipeline for knowledge compilation.</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>An NNF (also a DNNF and d-DNNF).</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>A sentence in FBDD and OBDD (both are drawn using a more compact notation than NNF notation).</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>A vtree and an SDD for $(A \land B) \lor (B \land C) \lor (C \land D)$.</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>A vtree, SDD, and OBDD for $(A \land B) \lor (C \land D)$.</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>An outline of the reduce-then-solve paradigm.</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>A vtree and a respecting DNNF.</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Distributing the clauses of {Y \lor \neg Z, Z \lor Q, \neg X \lor Z, X \lor \neg Y \lor Q} on a vtree. Internal nodes show assigned clauses.</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>A vtree, its cutset clauses, context clauses, and context variables, defined for the CNF {Y \lor \neg Z, Z \lor Q, \neg X \lor Z, X \lor \neg Y \lor Q}.</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>A vtree and a Decision-DNNF.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Distributing the clauses of CNF {Y \lor \neg Z, \neg X \lor Z, X \lor \neg Y, X \lor Q} over a vtree.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>The trace of an exhaustive DPLL.</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>An SDD and its negation (both respect the vtree in Figure 4.1(a)).</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>Eliminating order $X, Y, W, Z, Q$ on an undirected graph.</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>The primal, dual, and incidence graphs of the CNF $(X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)$.</td>
<td>68</td>
</tr>
</tbody>
</table>
5.3 A right-linear vtree induced by order \( X, Y_1, \ldots, Y_n \). Nodes \( v \) show \( CCNFs(\Gamma, Y) \) for \( \Delta_n = \{ X \lor Y_1, \ldots, X \lor Y_n, Y_1 \lor \ldots \lor Y_n \} \), where \( \Gamma \) is context clauses and \( Y \) is variables outside \( v \).

6.1 A vtree and a respecting Decision-SDD. .......................... 93
6.2 Macros for some SAT-solver primitives. .......................... 97
6.3 Comparison of minic2d against state-of-the-art model counters. 110

7.1 Forgetting variables \( X, Z \) from a d-DNNF. Decomposability is preserved but determinism is lost. ..................... 132
7.2 A jointree for CNF \( \Delta_n^b \) ........................................ 146

8.1 An SDD and a vtree for \( (A \land B) \lor (B \land C) \lor (C \land D) \). ....... 150
8.2 An \( X \)-constrained vtree, where \( X = \{ A, B, D \} \). ............... 152
8.3 Value registers computed by Algorithm 9. ......................... 154
8.4 Rotate a vtree node \( x \) right and left, and swap its children. Nodes \( a, b, \) and \( c \) may represent leaves or subtrees. ............... 161
8.5 General to \( X \)-constrained vtree, for \( X = \{ C, D \} \). ............... 163
List of Tables

2.1 Logical queries on a knowledge base $\Delta$. ........................................ 19

2.2 Languages and their polynomial time support to logical queries. $\circ$

means “does not support unless $P = NP$” and $?$ means “do not

know”. ............................................................................................................. 20

2.3 Instantiations of variables $A, B, C$ and whether they are a model

of CNF $\Delta = (A \lor B) \land (A \lor \neg C)$. X means not a model. ............. 27

5.1 The cutsets of CNF $(X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)$ with respect

to variable ordering $X, Y, Z, Q$. ............................................................. 69

5.2 Complexity parameters bounding the sizes of DNNF and its subsets. 76

6.1 Changes to adapt Algorithm 7 to do model counting. $weight(\ell)$

returns the weight of a literal $\ell$. ......................................................... 104

6.2 Changes to adapt Algorithm 7 to do weighted model counting.

$weight(\ell)$ returns the weight of a literal $\ell$. ....................................... 105

6.3 Bottom-up and top-down SDD compilations over iscas85, iscas89,

LGSynth89, and some sampled benchmarks. BU refers to bottom-up

compilation without dynamic minimization and BU+ with dynamic

minimization. TD refers to top-down compilation, and TD+ with a

single minimization step applied at the end. ......................................... 106

6.4 Comparison among model counters. An entry $(x, y)$ shows the num-

ber of instances solved by the system $x$ but not by the system $y$. 109

7.1 Experimental results on CNFs $\Delta_a^n$. c2d_forget is our approach

for compiling DNNFs without determinism. All timings are in sec-

onds. ............................................................................................................. 142
7.2 Some stats on CNF encodings of FPGA routing problems, before and after preprocessing using the BVA transformation. . . . . . . 143

7.3 Experimental results on FPGA routing problems. c2d_forget is our approach for compiling DNNFs without determinism. All timings are in seconds. . . . . . . . . . . . . . . . . . . . . . . . . . 143

8.1 Experimental results on DQMR-networks. . . . . . . . . . . . . 172
ACKNOWLEDGMENTS

Ph.D is a long and arduous journey which does not always end with one writing an acknowledgement section. I am lucky enough to write this section, and this would not have been possible without all my surroundings below.

I cannot imagine to have a better adviser than Adnan Darwiche. One would expect a student–adviser relationship to be quite a complex one, yet everything has been so simple with Adnan as he always supports and cares about his students. Adnan provided me with enough freedom to conduct my research while ensuring that I do not go out of track, which is quite common for a Ph.D student. I learned a lot from Adnan, all of which resulted in this dissertation and me being an independent researcher. Thanks Adnan!

Thanks to the members of my doctoral committee, Richard Korf, Todd Millstein, and Paulo Tabuada for the time they spent while serving on my committee. Their feedback and questions have improved the quality of this work. I would like to especially thank Rich for his various comments and edits on my dissertation, which have improved the readability.

Thanks to Arthur Choi, a research associate in our group, for being genuinely awesome. Sharing an office with him has been influential, inspiring, and fun in terms of everything. Thanks Arthur, you are a good friend!

Thanks to my friends from UCLA: Karthika Mohan, Vivek Menon, Suming Chen, Tiansheng Yao, Jason Shen, and Eunice Chen. I would not have survived in the gloomy corridors of Boelter Hall without you guys. I would like to especially thank Karthika for her constant support which helped me deal with many challenges.

Thanks to Esin Demirci Can, Dogan Can, and Tansel Uras for preventing me from being homesick.
Thanks to "Amcooollari", Inanc Arin, Ugur Bagci, Burak Ovant, Ersin Tannriverdi, and Yasar Tuzel for being a constant source of procrastination!

Thanks to Cleopatra VonLudwig, John Brady, Justin T. Hart, Benjamin Keil, Naka Pilantanakitti, and Anh Tran for teaching me "American Culture 101". I am already enrolled in "American Culture 102". You are the best slackers!

Thanks to my parents Gulcin Oztok and Necdet Oztok, and my lovely sister Basak Oztok for always being there for me. I simply have the best family!

Finally, there is Doga Gizem Kisa. I am very lucky to have Doga in my life. I do not think any words would be enough to describe her contribution to this dissertation and to me in general. As I guessed correctly back then, the grass is still greener with you, and will always be.
Vita

2005–2010 B.S. (Computer Science and Engineering), Sabanci University

2010–2012 M.S. (Computer Science and Engineering), Sabanci University

Publications


CHAPTER 1

Introduction

Knowledge compilation provides a general paradigm for dealing with computationally intractable problems (see, e.g., [Mar95, SK96, CD97, DM02, Dar14]). According to this paradigm, one first encodes a computational problem \( P \) in some input language, which can be viewed as modeling the problem. This encoding is then compiled into another tractable, output language, allowing instances of problem \( P \) to be answered efficiently in the new representation (i.e., in polynomial time). Under this paradigm, the computational burden that is intrinsic to the problem is pushed into an offline, compilation phase, with the goal of amortizing this cost over a large number of queries that can be answered efficiently on the compiled representation. Thus, one can deal with intractability in a broad and systematic way by devising efficient and general compilation algorithms, instead of developing algorithms and techniques that are tailored to specific problems.

For instance, consider the problem of product configuration, where the goal is to assemble a complex product according to users’ specifications. A concrete example here is the automotive industry, where customers would request to configure vehicles (e.g., a sedan with a rear view camera and a navigation system). As each customer would likely have different demands, it is not hard to imagine that a large number of configuration queries are asked by the customers of a car company. Moreover, it is known that deciding if a configuration is valid is a computationally hard task in general [MF89, KS00, FFJ04]. Yet, the set of valid vehicle configurations do not change much in time. Hence, one can employ knowledge compilation
in this area by compiling the configuration constraints into a language that allows one to efficiently check whether a particular car configuration is valid. While the compilation process may be computationally demanding, each individual configuration query would be efficient, which can lead to significant improvements in the overall runtime. Indeed, this approach has been implemented successfully in practice as knowledge compilation powers Toyota and Lexus configuration systems today.\(^1\)

Knowledge compilation has also been employed in the *reduce-then-solve* paradigm, which is exemplified by the use of SAT solvers when tackling problems in the complexity class \(\text{NP}\). According to this paradigm, one focuses their efforts on building a solver for a prototypical problem that is complete for a given complexity class. When tackling other problems in that class, one simply reduces these problems to the prototypical problem and applies the developed solver. Knowledge compilation was used in this regard to build solvers for prototypical problems that are complete for complexity classes that are harder and include the class \(\text{NP}\).

Given such advantages, naturally, knowledge compilation faces many challenges, from identifying tractable representations, to developing efficient compilation algorithms, to implementing them in effective knowledge compilers. This thesis targets various such challenges, and hence aims at expanding the use of knowledge compilation for solving computationally hard problems.

In particular, the thesis addresses the following main challenges:

- developing compilation methods that provide some theoretical guarantees on time and space complexities for various representations;

- building efficient knowledge compilers that advance the state-of-the-art;

- solving new Beyond-\(\text{NP}\) problems using knowledge compilation.

Our focus will be on the setting of propositional logic. In particular, we will target representations of Boolean functions, based on Boolean circuits that satisfy the property of decomposability. Such circuits create a spectrum of tractable representations: on one end there is decomposable negation normal form (DNNF) [Dar01a] being the most succinct and the least tractable one; on the other end there is ordered binary decision diagram (OBDD) [Bry86] being the least succinct and the most tractable one; in between there are several representations (see, e.g., [PD08]). Our interest will be on compiling intractable representations (e.g., CNFs) into representations in this spectrum. We next summarize our contributions on the challenges mentioned above.

Although compilation is often a hard task, parameterized knowledge compilation studies techniques that bound the time and space complexities of the compilation algorithm by some “parameter” of the given problem. Hence, given some representation that has a small such parameter, compilation would become easy. Thus far, different compilation methods targeting different representations have been introduced (see, e.g., [Dar01a, PD10a]). These methods are all parameterized by a well-known graph-theoretic property, called treewidth [RS84]. An interesting research direction is to identify new compilation methods parameterized by new parameters tighter than treewidth.

In this regard, we have introduced a new notion of width, called CV-width, that is specific to CNFs [OD14a]. The key insight is that CV-width takes the logical content of CNFs into account, whereas the classical notion of treewidth is only based on structural properties of CNFs. As such, CV-width strictly dominates the “incidence” treewidth of a CNF: it is no greater and can be bounded when the incidence treewidth is unbounded. We then developed a new compilation algorithm parameterized by CV-width that compiles CNFs into DNNFs. Therefore, we have significantly improved existing bounds on DNNF compilation.

Later, we proposed special cases of CV-width to parameterize the compilation
of subsets of DNNFs having the property of \textit{determinism}. The significance here is that those representations can be employed to efficiently solve model counting, which is key to probabilistic reasoning [Rot96]. We introduced \textit{decision-width} and developed an algorithm parameterized by this width that compiles CNFs into Decision-DNNFs [OD14b]. The new width strictly dominates the “primal” treewidth of a CNF; hence, we have improved on the best known upper bounds for compiling Decision-DNNFs. We also demonstrated that the output of the compilation algorithm can be converted in linear time to a sentential decision diagram (SDD) [Dar11], which led to a tighter upper bound on compiling SDDs. Further, we improved existing upper bounds on OBDD compilation by introducing \textit{linear} CV-width [OD14a]. In particular, we showed that linear CV-width strictly dominates both the “pathwidth” and “cutwidth” of a CNF, and provided an algorithm parameterized by linear CV-width that compiles CNFs into OBDDs.

While parameterized knowledge compilation provides some theoretical guarantees on compilation, developing efficient knowledge compilers, which is crucial to make knowledge compilation practical, remains a major challenge. Along this line, inspired by our algorithm parameterized by decision-width, we have developed a \textit{top-down} CNF-to-SDD compiler which integrates powerful techniques from the SAT literature [OD15]. When compared against the state-of-the-art, \textit{bottom-up} SDD compiler [CD13b], the new compiler shows orders-of-magnitude improvements in compilation time on certain benchmarks. Furthermore, the new compiler is based on a framework that would help facilitate the development of knowledge compilers and model counters [OD17a].

Another challenge to building efficient knowledge compilers concerns DNNF compilation. DNNF is one of the most succinct tractable representations that is suitable for various applications. Yet, there is no knowledge compiler that constructs a DNNF without enforcing another property, known as \textit{determinism}. This results in generating a more tractable but less succinct representation. Therefore,
the compilation task becomes harder (if feasible at all) unnecessarily for the applications that only require the property of decomposability. To deal with this matter, we introduced a new compilation algorithm to construct DNNFs without enforcing determinism [OD17b]. Our technique is based on a new notion of equivalence that makes use of auxiliary variables, and leverages existing compilers. Our experimental results show that the new method leads to significant improvements on certain benchmarks.

In addition to these, we have studied solving Beyond-\textbf{NP} problems using knowledge compilation. So far, this paradigm has been successfully employed to solve some \textbf{PP}-complete and \textbf{NP}^{\text{PP}}-complete problems for Bayesian networks. We have showed how knowledge compilation can be used to solve problems in the more intractable complexity class \textbf{PP}^{\text{PP}} [OCD16]. In particular, we identified a new property of SDDs that makes the prototypical \textbf{PP}^{\text{PP}}-complete problem \text{MajMajSat} tractable. To show the practical value of our approach, we adapted it to answer the \emph{same-decision probability} (SDP) query, which was recently introduced for Bayesian networks. The SDP problem, which is also \textbf{PP}^{\text{PP}}-complete, is a value-of-information query that quantifies the robustness of threshold-based decisions. We presented favorable empirical results, comparing our new approach with the state-of-the-art algorithm for computing the SDP.

We next provide an overview for the remaining chapters of this thesis.

In Chapter 2, we define our notation and give an introduction to knowledge compilation, placing emphasis on tractable representations of Boolean functions and solving Beyond-\textbf{NP} problems.

In Chapter 3, we propose CV-width for parameterizing CNF-to-DNNF compilation. This new notion of width is specific to CNFs and is motivated by a new decomposition technique that combines variable splitting and clause splitting. We introduce CV-width and show that CNFs can be compiled into DNNFs in time and space that are exponential only in CV-width.
In Chapter 4, we introduce special cases of CV-width for parameterizing the compilation of deterministic subsets of DNNF, assuming the input is a CNF. We introduce decision-width and show that CNFs can be compiled into both Decision-DNNFs and SDDs in time and space that are exponential only in this width. Furthermore, we introduce linear CV-width and show that CNFs can be compiled into OBDDs in time and space that are exponential only in linear CV-width.

In Chapter 5, we compare new notions of widths that we introduced with well-known graph abstractions of CNFs and their corresponding parameters. We show that CV-width strictly dominates the treewidth of the incidence CNF graph, decision-width strictly dominates the treewidth of the primal CNF graph, and linear CV-width strictly dominates the cutwidth and pathwidth of a CNF.

In Chapter 6, we introduce a new CNF-to-SDD compilation algorithm that integrates SAT technology, including clause learning, and corresponding open-source software. Our algorithm is based on a new framework that provides a modular design, a formal semantics, and a proof of correctness. Further, we empirically show that our compiler leads to significant improvements in compilation time against the state-of-the-art SDD compiler, assuming the input is a CNF. We also modify our algorithm to yield a new model counter, and perform an extensive empirical evaluation, comparing it to the state-of-the-art model counters.

In Chapter 7, we introduce a new DNNF compilation algorithm that constructs DNNFs without enforcing determinism. Our algorithm is based on a new type of equivalence that we introduce here. We show the applicability of our new method by presenting favorable empirical results against the existing DNNF compilers, which enforce determinism.

In Chapter 8, we extend the reach of knowledge compilation to problems in the highly intractable complexity class \( \text{PP}^{\text{PP}} \). We introduce a special class of SDDs, called \textit{constrained} SDDs, and develop an algorithm that solves the prototypical \( \text{PP}^{\text{PP}} \)-complete problem \textsc{MajMajSat} in linear time once the problem instance is
compiled into this new representation. We also modify our algorithm to solve a \( \text{PP}^{\text{PP}} \)-complete problem that is of practical interest, the same-decision probability (SDP) introduced for Bayesian networks. We empirically compare our proposed approach with the state-of-the-art algorithm for computing the SDP, and show favorable results. We conclude the thesis in Chapter 9.
CHAPTER 2

Technical Preliminaries

In this chapter, we introduce the basic notations, definitions, and concepts that will be consistently used throughout this thesis.

2.1 Propositional Logic

The foundation of the thesis is based on propositional logic, one of the most studied logic systems. We will provide a brief overview of its syntax and semantics.

2.1.1 Syntax

The syntax of a logic system is typically defined over an alphabet: symbols that are used to construct the basic elements known as sentences (i.e., formulas). The alphabet of propositional logic consists of propositional variables, constant symbols, logical operators, and parentheses. Here, propositional variables are those whose domain contains two constant values – true and false. Constant symbols are ⊤ and ⊥, denoting constant values true and false, respectively. Logical operators are ¬ (negation), ∧ (conjunction), and ∨ (disjunction). Given this alphabet, a propositional sentence, or simply a sentence, can be inductively defined as follows:

- each propositional variable \( X \) is a sentence;
- constant symbols \( \top \) and \( \bot \) are sentences;
- given a sentence \( \alpha \), \( \neg \alpha \) is a sentence;
• given two sentences $\alpha$ and $\beta$, $(\alpha \land \beta)$ is a sentence;

• given two sentences $\alpha$ and $\beta$, $(\alpha \lor \beta)$ is a sentence.

In other words, a sentence is formed by applying zero or more logical operators to propositional variables and constant symbols. Parentheses are used to prevent ambiguity, and can be often removed. For instance, the following are all sentences, where $X, Y,$ and $Z$ are propositional variables:

$$X, \neg X, \top, X \land Y, (X \land Y) \lor (\neg X \land Z).$$

A literal of a propositional variable is either the variable itself (e.g., $X$) or its negation (e.g., $\neg X$). Throughout this thesis, upper-case letters (e.g., $X$) will denote propositional variables and lower-case letters (e.g., $x$) will denote their instantiations. Bold upper-case letters (e.g., $X$) will denote sets of propositional variables and bold lower-case letters (e.g., $x$) will denote their instantiations.

### 2.1.2 Semantics

One common approach to define the semantics of propositional logic is through the notion of worlds. Consider a sentence $\alpha$ that is defined over propositional variables $X$. A world of sentence $\alpha$ is a mapping from variables $X$ to the constant values true and false. That is, a world instantiates variables of a sentence by assigning each one of them to one of the constant values true and false. Hence, a world is also known as a complete instantiation. Note that there are $2^{|X|}$ different possible worlds for a sentence defined over variables $X$. Throughout the thesis, we may liberally treat a world as the conjunction of literals that are mapped to true by the world. For instance, the conjunction $X \land Y \land \neg Z$ will represent the world $X = \text{true}, Y = \text{true}, Z = \text{false}.$

---

1. Another approach is to use truth tables, which yields the same semantics as ours.
Given the notion of worlds, we can evaluate a sentence to one of the constant values true and false. In particular, a world \( \omega \) satisfies a sentence \( \Delta \), denoted \( \omega \models \Delta \), according to the following definition:

- \( \omega \models X \) iff \( \omega \) maps variable \( X \) to true;
- \( \omega \models \top \);
- \( \omega \not\models \bot \);
- \( \omega \models (\neg \alpha) \) iff \( \omega \not\models \alpha \);
- \( \omega \models (\alpha \land \beta) \) iff \( \omega \models \alpha \) and \( \omega \models \beta \);
- \( \omega \models (\alpha \lor \beta) \) iff \( \omega \models \alpha \) or \( \omega \models \beta \).

If \( \omega \models \Delta \), then \( \Delta \) is evaluated to true at world \( \omega \). In this case, \( \omega \) is said to be a model of \( \Delta \). If \( \Delta \) has at least one model, then it is said to be satisfiable. The number of models of \( \Delta \), denoted \( \text{MC}(\Delta) \), is called its model count. As an example, consider the sentence \( \Delta = (X \land Y) \lor (\neg X \land Z) \). The world \( X = \text{true}, Y = \text{true}, Z = \text{false} \) satisfies sentence \( \Delta \) (i.e., it is a model of \( \Delta \)), whereas the world \( X = \text{false}, Y = \text{true}, Z = \text{true} \) does not. The model count of \( \Delta \) is four (i.e., \( \text{MC}(\Delta) = 4 \)), so that \( \Delta \) is satisfiable. On the other hand, the sentence \( X \land \neg X \) has no models, so it is unsatisfiable.

Now that we have defined the semantics of propositional logic, we will review some useful relationships and operations of sentences.

A sentence \( \alpha \) is logically equivalent to a sentence \( \beta \), denoted \( \alpha \equiv \beta \), iff \( \alpha \) and \( \beta \) have the same set of models. For instance, the sentences \( (X \land Y) \lor (X \land Z) \) and \( X \land (Y \lor Z) \) are logically equivalent. This is an important notion as it can be used to characterize whether two knowledge bases contain the same information.
A sentence $\alpha$ entails a sentence $\beta$, denoted $\alpha \models \beta$, iff each model of $\alpha$ is also a model of $\beta$. For instance, $(X \land Y) \lor (\neg X \land Z)$ entails $X \lor Z$. Entailment is an important relationship as it defines which information can be inferred from a knowledge base. In particular, if sentence $\alpha$ encodes a knowledge base that entails sentence $\beta$, then $\beta$ can be seen as a piece of information inferred from the knowledge base. This is because sentence $\alpha$ and sentence $\alpha \land \beta$ are logically equivalent when $\alpha \models \beta$. We finally note that $\bot \models \alpha$, $\alpha \models \top$, and $\alpha \models \alpha$ for any arbitrary sentence $\alpha$.

The conditioning of a sentence $\alpha$ on an instantiation $z$ of variables $Z$, denoted $\alpha|z$, is to replace variables $Z$ in $\alpha$ with constant symbols $\top$ and $\bot$ corresponding to their values in instantiation $z$. For example, conditioning the sentence $(X \land Y) \lor (\neg X \land Z)$ on the instantiation $X = \text{false}$, $Y = \text{true}$ results in the sentence $(\bot \land \top) \lor (\neg \bot \land Z)$. One often simplifies the sentence after conditioning. For instance, the above sentence simplifies to sentence $Z$. Conditioning is a useful operation as it allows one to update a knowledge base when some evidence is obtained regarding variables.

The existential quantification of a variable $X$ from a sentence $\alpha$, denoted $\exists X. \alpha$, is the sentence obtained by disjoining sentences $\alpha|X$ and $\alpha|\neg X$ (that is, $\exists X. \alpha = \alpha|X \lor \alpha|\neg X$). Existential quantification is also known as forgetting since the resulted sentence does not mention the quantified variable (i.e., the variable is forgotten). One can also existentially quantify a set $X$ of variables by successively quantifying variables in $X$.

---

2We overload the notation $\models$. Yet, this usage is consistent with the previous one when a world is interpreted as the conjunction of literals that are mapped to true by the world.
2.2 Boolean Functions

An important concept that we will use throughout the thesis is Boolean functions. A Boolean function \( f \) over a set \( Z \) of propositional variables (simply, \( f(Z) \)) is a function that maps each instantiation \( z \) of variables \( Z \) to one of the constant values true and false. For instance, the odd parity function over a set of variables is defined to map an instantiation to true iff an odd number of variables is set to true in the instantiation. A trivial Boolean function maps all possible instantiations to true (denoted \( \top \)) or maps them all to false (denoted \( \bot \)).\(^3\) An instantiation \( z \) satisfies Boolean function \( f(Z) \), denoted \( z \models f \), iff \( f \) maps \( z \) to true. Given this notion, all related properties and relationships regarding (propositional) sentences directly apply to Boolean functions (e.g., satisfiability and entailment).

The conditioning of a Boolean function \( f \) on an instantiation \( z \), denoted \( f|z \), is the sub-function obtained by setting variables \( Z \) to their values in \( z \).\(^4\) Note that the sub-function does not depend on the conditioned variables. For example, let \( f \) be the odd parity function over variables \( X,Y,Z \). The conditioning of \( f \) on the instantiation \( X = \text{true} \), \( f|X = \text{true} \), results in the even parity function over variables \( Y,Z \) (i.e., maps to true whenever an even number of variables is set to true). We may sometimes refer to a Boolean function as simply a function.

Indeed, each Boolean function can always be expressed as a sentence. One simple way to do this is to disjoin the models of a Boolean function (treating each model as a conjunction of literals). For instance, the odd parity function over variables \( X,Y,Z \) can be represented using the following sentence (which is simply constructed by disjoining the models):

\[
(X \land \neg Y \land \neg Z) \lor (\neg X \land Y \land \neg Z) \lor (\neg X \land \neg Y \land Z) \lor (X \land Y \land Z).
\]

In fact, a Boolean function may be represented by different sentences. For in-

\(^3\)The usage of symbols \( \top \) and \( \bot \) is consistent with those used in propositional sentences.
\(^4\)This definition generalizes the conditioning of propositional sentences.
stance, the above function can also be represented using the following sentence:

\[(\neg X \lor \neg Y \lor Z) \land (\neg X \lor Y \lor \neg Z) \land (X \lor \neg Y \lor \neg Z) \land (X \lor Y \lor Z).\]

However, we will not assume any specific form when we refer to a Boolean function.\(^5\) Yet, we will next show a particular form that is commonly used to express Boolean functions.

### 2.2.1 Conjunctive Normal Form

A common way to express Boolean functions is based on propositional forms. One such popular form, which is suitable for human specification and interpretation, is conjunctive normal form. A sentence in *conjunctive normal form* (CNF) is a conjunction of clauses, where each clause is a disjunction of literals. For instance, \(\Delta = (X \lor \neg Y \lor \neg Z) \land (Y \lor Z) \land \neg X\) is in CNF. Indeed, \(\Delta\) has three clauses with the last one consisting of a single literal. A clause with a single literal is called a *unit* clause. A clause can also be empty (i.e., has no literals in it), in which case it is equivalent to \(\bot\). Similarly, a sentence in CNF can be empty (i.e., has no clauses in it), in which case it is equivalent to \(\top\). These last two properties hold as \(\bot\) is the identity of the disjunction operator (i.e., \(\lor\)) and \(\top\) is the identity of the conjunction operator (i.e., \(\land\)). We will often refer to a sentence in CNF as simply a CNF.

Conditioning a CNF can be performed easily, without changing the form of the conditioned sentence (i.e., keeping it as a CNF). Basically, conditioning a CNF \(\Delta\) on a literal \(\ell\) amounts to removing literal \(\neg \ell\) from all clauses and then dropping all clauses that contain literal \(\ell\). For instance, conditioning \((X \lor \neg Y \lor \neg Z) \land (Y \lor Z) \land \neg X\) on literal \(Z\) results in the sentence \((X \lor \neg Y) \land \neg X\), which is a CNF. This can be generalized to instantiations (i.e., sets of literals) straightforwardly: one keeps repeating the same process for each literal of the given instantiation.

\(^5\)Regardless of this, we may sometimes combine Boolean functions using the traditional logical operators, such as \(\land, \lor, \oplus\), and \(\leftrightarrow\).
It is not uncommon to represent CNFs in set notation: a CNF can be seen as a set of clauses, where each clause is a set of literals. For instance, CNF \((X \lor \neg Y \lor \neg Z) \land (Y \lor Z) \land \neg X\) can be represented as \(\{X, \neg Y, \neg Z\}, \{Y, Z\}, \{\neg X\}\). We may sometimes use this notation to represent CNFs.

### 2.3 Knowledge Compilation

We will now review knowledge compilation, which is the main theme of this thesis. A typical pipeline of how this paradigm works is given in Figure 2.1. Here, the main goal is to efficiently answer certain queries on a given knowledge base. However, these queries are often intractable with respect to the initial representation of the knowledge base. To deal with this intractability, we can identify a target language that supports the necessary operations in polynomial time, and then construct a tractable representation of the knowledge base in the chosen language. Once the construction is completed, we can perform the required reasoning tasks efficiently. Indeed, the compilation process would be computationally costly (e.g., exponential time). Yet, the goal is to amortize this cost through answering many queries efficiently, and hence making the overall process more efficient than without the compilation.

Although it explains the basics, the pipeline above only provides an abstract view of knowledge compilation. To make it more concrete, we will discuss the main components of the pipeline: input languages, target languages, and properties of languages.

Input languages are often those that can easily encode the knowledge base. Although encoding knowledge bases is out of the scope of this thesis, one common choice here is CNF as it is often convenient to describe a system as a conjunction of its different components. Target languages and their properties, on the other hand, is one of the main research topics in knowledge compilation. Hence, we will
provide an overview of these topics.

### 2.3.1 Target Languages

The basis of target representations that we will study is a general language known as negation normal form (NNF) [Dar01a]. A (propositional) sentence is in NNF iff the negation operator ($\neg$) only occurs immediately in front of variables. For instance, the following is a sentence in NNF:

$$\Delta = (\neg X \land ((Y \land \neg Z) \lor (\neg Y \land Z))) \lor (X \land \neg Y \land \neg Z \land Q).$$

Indeed, each sentence can be written in NNF: one simply needs to keep applying the DeMorgan law until the negation operator is only applied to variables. Hence, NNF is a complete language. For instance, the sentence $X \land \neg(Y \lor Z)$ is not in NNF. Yet, we can transform it into a sentence in NNF by moving the negation inwards, which would result in the sentence $X \land (\neg Y \land \neg Z)$.

Moreover, each sentence in NNF can be represented using a rooted, directed acyclic graph (DAG) whose leaf nodes are labeled with either literals or constant symbols ($\top$ and $\bot$) and whose internal nodes are labeled with either conjunction ($\land$) or disjunction ($\lor$). For instance, Figure 2.2 depicts the graph representation of the sentence $\Delta$ given above. In the rest of the thesis, we will always use this graph.
representation when referring to a sentence in NNF and will make the relevant definitions and notions on this graph structure. We will often refer to a sentence in NNF as simply an NNF (this also applies to subsets of NNF discussed throughout the thesis).

Given an NNF node $\alpha$, we will use $\text{Vars}(\alpha)$ to denote the set of variables appearing in the DAG rooted at $\alpha$. We will often not distinguish between an NNF node $\alpha$ and the DAG rooted at node $\alpha$. We will also identify an NNF by its root node. The size of an NNF $\alpha$, denoted $|\alpha|$, will refer to the number of edges of the DAG rooted at $\alpha$.

We note that an NNF is not an interesting representation as is. However, by imposing certain properties on the structure of NNFs, one can obtain tractable representations. We will consider next some fundamental properties that lead to influential languages in knowledge compilation.

Decomposability is a property of conjunction nodes [Dar01a]. A conjunction node is \textit{decomposable} iff its children do not share variables. That is, if $C_1, \ldots, C_n$ are the children of a conjunction node, then decomposability implies that $\text{Vars}(C_i) \cap \text{Vars}(C_j) = \emptyset$ for $i \neq j$. This property leads to the \textit{decomposable}
negation normal form (DNNF) language, which is the set of NNFs whose conjunction nodes are decomposable. For instance, Figure 2.2 also depicts a DNNF.

Determinism is a property of disjunction nodes [Dar01b]. A disjunction node is deterministic iff each pair of its children are logically inconsistent (i.e., mutually exclusive). That is, if $C_1, \ldots, C_n$ are the children of a disjunction node, then determinism implies that $C_i \land C_j$ is unsatisfiable for $i \neq j$. Determinism and decomposability lead to the deterministic decomposable negation normal form (d-DNNF) language, which is the set of NNFs whose disjunction nodes are deterministic and whose conjunction nodes are decomposable. So, d-DNNF is a subset of DNNF. For instance, Figure 2.2 also shows a d-DNNF.

Another important property is the notion of decision nodes, which is recursively defined as follows [Bry86]. An NNF node $N$ is called a decision node iff $N$ is labeled with $\top$, $\bot$, or is a disjunction node having the form $(X \land \alpha) \lor (\neg X \land \beta)$ where $\alpha$ and $\beta$ are decision nodes and $X$ is a propositional variable. Decision nodes and decomposability lead to the free binary decision diagram (FBDD) language, which is the set of NNFs whose roots are decision nodes and whose conjunction nodes are decomposable. Clearly, decision nodes are already deterministic; hence FBDD is a subset of d-DNNF. Instead of using the NNF notation, FBDDs are typically drawn using the following more compact notation:

Here, a solid edge refers to $X$ being true (pointing to the high child) and a dashed edge refers to $X$ being false (pointing to the low child). Figure 2.3(a) depicts an FBDD using this more compact notation.

A closer look at the more compact notation for FBDDs will reveal that each
variable appears at most once on each path from the root to a leaf. Ensuring that variables also appear in the same order on each root-to-leaf path leads to the influential ordered binary decision diagram (OBDD) language [Bry86]. That is, OBDD is a subset of FBDD that satisfies the ordering property. For a given variable ordering $\prec$, we will use $\text{OBDD}_\prec$ to denote the set of OBDDs respecting $\prec$. Figure 2.3(b) depicts an OBDD respecting the variable order $A, B, C, D$.

Before studying some properties of the languages discussed so far, we note that CNF can also be defined as a subset of NNF, where the height of the corresponding DAG is at most two and the children of any disjunction node are leaves (i.e., representing a clause).

### 2.3.2 Tractability of Representations

Now that we have defined some target languages, we will turn our focus to their tractability, which refers to the set of queries that have polynomial time implementations on a given target language. In Table 2.1, we show some well-known queries that are used in practice. For instance, model counting (CT) is key to

---

6As such, FBDDs are also known as read-once branching programs in the theory literature.
probabilistic inference and equivalence checking (EQ) is crucial in formal verification. In Table 2.2, we present some known results regarding the queries in Table 2.1 [DM02].

Clearly, NNF is the least tractable language and OBDD< is the most tractable language. In fact, only looking at Table 2.2 will show us that OBDD< is the best choice all the time. However, tractability comes with a price: the more tractable a language is, the less succinct it will be. That is, it will be harder to compile into an OBDD< without blowing up the size of the representation, compared to other less tractable representations. Since query answering takes polynomial time in the size of the representation, we need to find a balance between tractability and succinctness while choosing a language for a specific reasoning task. Therefore, we will next review the succinctness property in more detail.

### 2.3.3 Succinctness of Representations

Succinctness allows us to compare the sizes of target representations. More formally, we will use the following definition taken from [DM02]. Consider two subsets $L_1$ and $L_2$ of NNF. $L_1$ is at least as succinct as $L_2$, denoted $L_1 \leq L_2$, iff there exists a polynomial $p$ such that for every sentence $\beta$ in $L_2$, there exists a
Table 2.2: Languages and their polynomial time support to logical queries. ◦ means “does not support unless $P = NP$” and ? means “do not know”.

logically equivalent sentence $\alpha$ in $L_1$ where the size of $\alpha$ is bounded by the size of $\beta$ in terms of the polynomial $p$ (i.e., $|\alpha| \leq p(|\beta|)$). Furthermore, $L_1$ is *strictly more succinct* than $L_2$, denoted $L_1 < L_2$, iff $L_1 \leq L_2$ and $L_2 \nleq L_1$. For instance, $\text{FBDD} \leq \text{OBDD}$ as OBDD is a subset of FBDD. Also, [GM94] showed a Boolean function that has a polynomial size FBDD but only exponential size OBDDs. So, $\text{OBDD} \nleq \text{FBDD}$, implying that $\text{FBDD} < \text{OBDD}$. In this case, it is common to say that FBDDs are exponentially more succinct than OBDDs.

Accordingly, we have the following succinctness relationship between the languages presented in Table 2.2:

\[
\text{NNF} < \text{DNNF} < \text{d-DNNF} < \text{FBDD} < \text{OBDD} < \text{OBDD}_<.
\]

As expected, $\text{OBDD}_<$ is the least succinct language. Still, it supports some polynomial time queries that are not supported by others (e.g., equivalence checking). Hence, identifying more succinct languages than OBDD, without compromising tractability, has been a crucial goal in knowledge compilation. We will next study one of the attempts made in this direction, which leads to a representation that is essentially as tractable as OBDDs while being exponentially more succinct.
2.3.4 Sentential Decision Diagram (SDD)

SDD is a subset of d-DNNF in which stronger versions of decomposability and determinism are imposed on the underlying NNF. The fundamental concept here is partitioning a function. Consider a Boolean function $f$ over variables $Z$. Let $X$ and $Y$ be two sets of variables that partition $Z$ (i.e., $X \cap Y = \emptyset$ and $X \cup Y = Z$). Then, function $f$ can always be decomposed into the following form [Dar11]:

$$f(Z) \equiv (p_1(X) \land s_1(Y)) \lor \ldots \lor (p_n(X) \land s_n(Y))$$

such that the sub-functions $p_i(X)$ are consistent, mutually exclusive, and exhaustive. That is, $p_i \not\equiv \bot$ for all $i$; $p_i \land p_j \equiv \bot$ for $i \neq j$; and $\lor_i p_i \equiv \top$. A decomposition of this kind is called an $(X, Y)$-partition, and denoted $\{(p_1, s_1), \ldots, (p_n, s_n)\}$.

In this case, each $p_i$ is called a prime and each $s_i$ is called a sub. Indeed, an $(X, Y)$-partition imposes both decomposability and determinism. Moreover, an $(X, Y)$-partition is compressed when its subs are distinct (i.e., $s_i \not\equiv s_j$ for $i \neq j$).

For instance, consider the Boolean function $f = (A \land B) \lor (B \land C) \lor (C \land D)$. Choosing $X = \{A, B\}$ and $Y = \{C, D\}$ yields the following $(X, Y)$-partition:

$$\{(\underbrace{A \land B}_\text{prime}, \top), (\underbrace{\neg A \land B}_\text{prime}, C), (\underbrace{\neg B}_\text{prime}, \underbrace{D \land C}_\text{sub})\}.$$

SDDs will result from the recursive decomposition of a Boolean function using $(X, Y)$-partitions. This recursive decomposition process requires a structure to determine the $X/Y$ variables of each partition. This is achieved by using an auxiliary structure called a variable tree [PD08]. A variable tree (vtree) for variables $X$ is a full binary tree whose leaves are labeled with variables $X$. For example, Figure 2.4(a) depicts a vtree for variables $A, B, C, D$. For an internal vtree node $v$, we will use $v^l$ and $v^r$ to denote the left and right children of $v$. We will denote variables appearing at or below the subtree rooted at vtree node $v$ by $Vars(v)$. We will also not distinguish between a vtree node $v$ and the subtree rooted at $v$.

We are now ready to formally define SDDs [Dar11]:
Definition 1. \( \alpha \) is an SDD that respects a vtree \( v \) iff:

- \( \alpha = \bot \) or \( \alpha = \top \);
- \( \alpha = X \) or \( \alpha = \neg X \) and \( v \) is a leaf node labeled with \( X \);
- \( \alpha = \{(p_1,s_1),\ldots,(p_n,s_n)\} \), \( v \) is an internal node, \( p_1,\ldots,p_n \) are SDDs that respect subtrees of \( v^l \), and \( s_1,\ldots,s_n \) are SDDs that respect subtrees of \( v^r \).

A constant or literal SDD is called a terminal; otherwise it is called a decomposition. SDDs will be depicted graphically as in Figure 2.4(b). Here, each decomposition is represented by a circle, where each element is depicted by a paired box \([p\,s]\). The left box corresponds to a prime \( p \) and the right box corresponds to its sub \( s \). A prime \( p \) or sub \( s \) are either a constant, literal, or pointer to a decomposition SDD. Decomposition SDDs are labeled with the vtree nodes they respect. This depiction can easily be converted into a traditional NNF notation: we replace circle-nodes with \( \lor \) and paired boxes \([p\,s]\) with \( p \land s \).

An SDD may contain trivial decomposition nodes which correspond to \((X,Y)\)-partitions of the form \{\((\top,\alpha)\)\} or \{\((\alpha,\top), (\neg\alpha, \bot)\)\}. When these decomposition nodes are removed (by directing their parents to \( \alpha \)), the resulting SDD is trimmed.

Figure 2.4: A vtree and an SDD for \((A \land B) \lor (B \land C) \lor (C \land D)\).
Moreover, an SDD is *compressed* when each of its partitions is compressed. For instance, the SDD in Figure 2.4(b) is indeed compressed and trimmed. For a fixed vtree, a Boolean function has a unique SDD that is compressed and trimmed. Hence, compressed and trimmed SDDs are canonical. Unless otherwise stated, we restrict our attention to compressed and trimmed SDDs throughout the thesis.

Indeed, an OBDD is a compressed and trimmed SDD that respects a special type of vtree, called right-linear.\(^7\) A vtree is *right-linear* when each left child is a leaf. In this case, the underlying \((X, Y)\)-partitions of the SDD will be of the form \(\{(X, \alpha), (\neg X, \beta)\}\), which corresponds to the *Shannon* decomposition used in OBDDs. Thus, SDDs contain OBDDs. Figure 2.5 depicts a right-linear vtree, an SDD that respects the same vtree, and the corresponding OBDD representation of the SDD.

We note that OBDDs base their decisions on single literals, whereas general

\(^7\)More precisely, *reduced* OBDDs [Bry86] correspond to compressed and trimmed SDDs, which can be constructed from OBDDs using some syntactic rules preserving polynomial size.
SDDs base their decisions on sentences. This generalization makes SDDs exponentially more succinct than OBDDs, as [Bov16] identified a class of Boolean functions that have polynomial size SDDs but only exponential size OBDDs. Due to this and the fact that SDDs contain OBDDs, SDDs are strictly more succinct than OBDDs.

2.4 Beyond NP

We will now present a perspective on this thesis from the broader context of solving Beyond-NP problems using knowledge compilation.

The reduce-then-solve paradigm, outlined in Figure 2.6, is based on the following property: a computational problem \( \mathcal{P} \) is complete for a complexity class \( \mathcal{C} \) if \( \mathcal{P} \) belongs to \( \mathcal{C} \) and every other problem in \( \mathcal{C} \) can be reduced to \( \mathcal{P} \) in polynomial time. This strong property has been commonly used to suggest that efficiently solving one complete problem in a complexity class would lead to efficiently solving all other problems within the same complexity class through polynomial time reductions. In fact, the efforts and investment made on SAT solvers has proven that this property can also be exploited in practice for NP-complete problems.

Indeed, there are other complexity classes beyond NP. Under some standard complexity-theoretic assumptions, problems complete for these classes are harder than ones for the class NP. Still, there are important practical problems that belong to classes beyond NP. Hence, extending the reduce-then-solve paradigm to Beyond-NP problems is a challenging and rewarding task.

Although the original motivation of knowledge compilation is to push as much of the computational effort into the compilation phase and then later amortize it over answering many on-line queries, the developments made so far allows this paradigm to be used for solving beyond NP problems in a systematic manner. The key here is to identify prototypical complete problems for complexity classes along
Problem $X$

Figure 2.6: An outline of the reduce-then-solve paradigm.

with the corresponding tractable NNF representations. Solution to a complete problem is then achieved by reduction to the representative prototypical problem and the use of the corresponding knowledge compiler. We will next briefly review how knowledge compilation has been used as a part of the reduce-then-solve paradigm for problems arising in probabilistic inference.

We will start with identifying some beyond NP classes that are relevant to probabilistic computations. For standard definitions from complexity theory, we refer the reader to [Pap94]. Our focus will be on the complexity classes NP, PP, NP$^{PP}$, and PP$^{PP}$, which are related in the following way:

$$\text{NP} \subseteq \text{PP} \subseteq \text{NP}^{PP} \subseteq \text{PP}^{PP}.$$ 

The class NP is the set of decision problems that can be solved by a non-deterministic polynomial time Turing machine. The class PP is the set of decision problems that can be solved by a non-deterministic polynomial time Turing machine, which has more accepting than rejecting paths. NP$^{PP}$ and PP$^{PP}$ are the corresponding classes
assuming a PP oracle. That is, the corresponding Turing machine has an access to a PP oracle.

The above complexity classes are crucial for probabilistic inference, as many probabilistic reasoning problems are complete for them. For instance, the most probable explanation (MPE) problem is NP-complete [Shi94]; computing the marginal probabilities is PP-complete [Rot96]; the maximum a posteriori hypothesis (MAP) problem is \( \text{NP}^\text{PP} \)-complete [Par02]; and the same-decision probability (SDP) problem is \( \text{PP}^\text{PP} \)-complete [CXD12].

To apply the reduce-then-solve-paradigm on these problems, we need to identify prototypical problems for the corresponding complexity classes. Those problems will be defined on CNF representations of Boolean functions. In particular, given a CNF \( \Delta \) representing Boolean function \( f(X, Y) \), we have the following decision problems which are respectively complete for the above complexity classes.

**Sat** is the prototypical NP-complete problem [Coo71], asking if there is an instantiation \( xy \) that satisfies CNF \( \Delta \).

**MajSat** is the prototypical PP-complete problem [Gil77], asking if there are a majority of instantiations \( xy \) that satisfy CNF \( \Delta \).

**E-MajSat** is the prototypical \( \text{NP}^\text{PP} \)-complete problem [LGM98], asking if there is an instantiation \( x \) for which a majority of instantiations \( y \) satisfy CNF \( \Delta|x \).

**MajMajSat** is the prototypical \( \text{PP}^\text{PP} \)-complete problem [Wag86], asking if there are a majority of instantiations \( x \) for which a majority of instantiations \( y \) satisfy CNF \( \Delta|x \).

As an example, consider the CNF \( \Delta = (A \lor B) \land (A \lor \neg C) \), which is defined over variables \( A, B, C \). In Table 2.3, we present all possible (complete) instantiations of CNF \( \Delta \), denoting whether an instantiation is a model in the last column. Accordingly, CNF \( \Delta \) has 5 models (i.e., satisfying instantiations), which is more
Table 2.3: Instantiations of variables $A, B, C$ and whether they are a model of CNF $\Delta = (A \lor B) \land (A \lor \neg C)$. $X$ means not a model.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>✓</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>✓</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>✓</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>✓</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>✓</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>X</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>X</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>X</td>
</tr>
</tbody>
</table>

than the half of all instantiations. Then, the answer to both $\text{Sat}$ and $\text{MajSat}$ is a “yes”. Now assume that we split the variables of $\Delta$ into two disjoint sets of variables $X$ and $Y$, where $X = \{A\}$ and $Y = \{B, C\}$. Then, the answer to $\text{E-MajSat}$ is a “yes” (due to the instantiation $A = \text{true}$), whereas the answer to $\text{MajMajSat}$ is a “no”.

Knowledge compilation presents a systematic approach for solving the above prototypical problems. In particular, those problems become tractable when the Boolean functions are compiled into an appropriate subset of NNF: $\text{Sat}$ is tractable on DNNFs [Dar01a]; $\text{MajSat}$ is tractable on d-DNNFs [Dar01b]; $\text{E-MajSat}$ is tractable on d-DNNFs where certain variables of interest appear in certain positions [HCD06]; and $\text{MajMajSat}$ is tractable on a subset of SDDs that we identified [OCD16].

The last step is to provide reductions to the prototypical problems from the mentioned probabilistic reasoning problems, which are all defined on Bayesian networks.\(^8\) This is achieved by capturing the probability distributions of a Bayesian

\(^8\)We refer the reader to [Dar09] for an introduction to Bayesian networks.
network as a CNF [Dar02] (see [SBK05, CD08, CKD13] for further practical approaches and reviews of encoding Bayesian networks into CNFs). Compiling this CNF encoding into the corresponding NNF class will then render the problems tractable.

Indeed, this systematic approach has been studied before on the mentioned Beyond-NP problems arising from probabilistic reasoning: for computing the marginal probabilities [Dar02], the MAP [HCD06], and recently the SDP [OCD16]. All of these results put knowledge compilation at the center of solving Beyond-NP problems through the reduce-then-solve paradigm.
CHAPTER 3

CV-Width: A New Complexity Parameter for CNFs

In this chapter, we propose a new notion of complexity parameter, called clause-variable width (CV-width), to parameterize the compilation of CNFs into DNNFs. This new notion of parameter is specific to CNFs and takes into account logical content of the CNF through a new decomposition technique that combines variable splitting and clause splitting. We introduce CV-width and show that CNFs can be compiled into DNNFs in time and space that are exponential only in CV-width. The material in this chapter is based on the work published in [OD14a].

3.1 Introduction

We have presented many tractable representations in Chapter 2 that share one fundamental property, called decomposability. As such, compiling knowledge bases (e.g., CNFs) into decomposable representations has been at the center of attention in the area of knowledge compilation. A key interest here is in providing upper bounds on the complexity of compilation algorithms (see, e.g., [Dar01a, HD04, PD10a, Dar11, RP13]).

So far, these upper bounds have been based on some structural properties of the input CNF. A common choice is to employ various graph abstractions of CNFs with the well-known graph-theoretic parameter, called treewidth [RS84].\(^1\) As an

\(^1\)Intuitively, treewidth can be seen as a parameter of a graph that measures its closeness to
example, consider the primal graph of a CNF: a graph where nodes are labeled with CNF variables and edges connect variables appearing in the same clauses. In this case, we can compile a CNF with \( n \) variables into a DNNF of size \( O(n^{2w}) \) where \( w \) is the treewidth of the CNF primal graph. Hence, compiling a DNNF can be done efficiently when the input CNF has small treewidth.

However, considering only structural properties of CNFs leads to losing information regarding the logical content of the CNF. For instance, two logically different CNFs might have the same primal graph abstraction, and hence will have the same upper bound mentioned above. Yet, this could lead to generate loose upper bounds as logical content can also significantly effect the compilation size.

Motivated by this, we introduce a new notion of width, called CV-width, which is a complexity parameter for CNFs that also takes logical content into account. We show that CNFs can be compiled into structured DNNFs [PD08] in time and space that are exponential only in CV-width. The significance here is that structured DNNF supports a polynomial time conjoin operation [PD08], while (unstructured) DNNF does not support this (unless \( P=NP \) ) [DM02]. As we will show later (in Chapter 5), CV-width strictly dominates the incidence treewidth of a CNF; hence, not only does our result improve on the best known bound for compiling DNNFs [RP13], but it also extends the bound to structured DNNF.

Our complexity result is constructive as it is based on a specific algorithm for compiling CNFs into structured DNNFs. This algorithm is driven by a vtree over CNF variables. Accordingly, each vtree has its own CV-width and the CV-width of a given CNF is the smallest width attained by any of its vtrees. The major characteristic of this algorithm is its employment of both variable and clause splitting. Variable splitting is a well-known technique in both SAT and knowledge compilation and calls for eliminating a variable \( V \) from a CNF \( \Delta \) by considering the CNFs \( \Delta|V \) and \( \Delta|\neg V \) (i.e., conditioning \( \Delta \) on both phases of a tree structure. For instance, the more connected a graph is the higher treewidth it will have.
the variable). Clause splitting, however, is a less common technique and calls for eliminating a clause $\alpha \lor \beta$ from a CNF $\Delta$ by considering the CNFs $\Delta \land \alpha$ and $\Delta \land \beta$.

Our proposed algorithm combines both techniques. This combination is essential for the complexity of our compilation algorithm and provides the major insight underlying the new notion of CV-width. Moreover, the combination allows us to bound the complexity of compilation in situations where this complexity could not be bounded using either technique alone.

This chapter is structured as follows. We start by providing some technical preliminaries and formal definitions of variable and clause splitting (Sections 3.2–3.5). This is followed by presenting our compilation algorithm (Section 3.6). Then, we introduce CV-width and show that our compilation algorithm is parameterized by this new notion of width (Section 3.7). We close the chapter by presenting the proofs of technical results.

### 3.2 Structured DNNF

In this section, we will review the notion of structuredness in DNNFs [PD08]. A DNNF respects a vtree iff every conjunction node $N$ of the DNNF has exactly two children $N^l$ and $N^r$ and we have $\text{Vars}(N^l) \subseteq \text{Vars}(v^l)$ and $\text{Vars}(N^r) \subseteq \text{Vars}(v^r)$ for some vtree node $v$. In this case, the DNNF is said to be structured. For instance, the DNNF in Figure 3.1(b) respects the vtree in Figure 3.1(a) and is therefore a structured DNNF. Indeed, SDDs and OBDDs are both a subset of structured DNNFs with stronger properties.

### 3.3 Decomposing CNFs

We will now describe a general method to construct structured DNNFs from CNFs. Consider a vtree with root $v$. Let $X$ be the variables of left child $v^l$ and
let \( Y \) be the variables of right child \( v^r \). To compile a CNF \( \Delta \) into a DNNF that respects this vtree, we will first decompose \( \Delta \) into CNFs (called components) that only mention variables \( X \) or only mention variables \( Y \). These components are then decomposed with respect to the vtrees rooted at \( v^l \) and \( v^r \). The process continues recursively until we reach literals or constants. The following definition provides the basis for this recursive decomposition process.

**Definition 2** ([PD10b]). Consider a CNF \( \Delta(X, Y) \) defined over disjoint sets of variables \( X \) and \( Y \). An \((X, Y)\)-decomposition of CNF \( \Delta \) is a set

\[
\left\{ \left( L_1(X), R_1(Y) \right), \ldots, \left( L_n(X), R_n(Y) \right) \right\}
\]

such that \( L_i \) and \( R_i \) are CNFs and \( \Delta \) is equivalent to \( (L_1 \land R_1) \lor \ldots \lor (L_n \land R_n) \). Each pair \((L_i, R_i)\) is called an element, where \( L_i \) is called an \( X \)-component and \( R_i \) is called a \( Y \)-component.

Consider the CNF \( \Delta = \{A \lor \neg B \lor \neg C, \neg A \lor B \lor C\} \) and let \( X = \{A, B\} \) and \( Y = \{C\} \). The following is then an \((X, Y)\)-decomposition of CNF \( \Delta \), which has three elements:

\[
\left\{ \left\{ \{A \lor \neg B, \neg A \lor B\}, \{} \right\}, \left\{ \{A \lor \neg B\}, \{C\} \right\}, \left\{ \neg A \lor B, \neg C \right\} \right\}.
\]
That is, CNF $\Delta$ is equivalent to the following sentence:

$$\left( (A \lor \neg B) \land (\neg A \lor B) \land \top \right) \lor \left( (A \lor \neg B) \land C \right) \lor \left( (\neg A \lor B) \land \neg C \right).$$

### 3.4 Constructing Decompositions

We will now review two systematic methods for constructing ($X$, $Y$)-decompositions. The first method is based on variable splitting [Dar01a] and the second one is based on clause splitting [PD10a].

#### 3.4.1 Decomposition by Splitting on Variables

To split on variables $V$ is to consider all possible instantiations $v$ of these variables. Here, each instantiation $v$ corresponds to a set of literals, exactly one literal for each variable in $V$. Hence, if $V$ contains $n$ variables, then splitting on variables $V$ leads to $2^n$ cases.

Consider now a CNF $\Delta$ defined over disjoint sets of variables $X$ and $Y$. Suppose further that the CNF is partitioned into three components: $\Delta(X)$, $\Delta(Y)$ and $\Delta(X,Y)$, where $\Delta(X)$ contains all clauses of $\Delta$ that only mention variables $X$ and $\Delta(Y)$ contains all clauses of $\Delta$ that only mention variables $Y$. Let $V$ be all variables in $X$ that are mentioned in $\Delta(X,Y)$. The following is then an ($X$, $Y$)-decomposition of CNF $\Delta$ [Dar01a]:

$$\left\{ \left( v \cup \Delta(X)|v, \Delta(Y) \cup \Delta(X,Y)|v \right) \mid v \text{ an instantiation of } V \right\}.$$  

This implies that

$$\Delta = \bigvee_v \left( v \land (\Delta|v) \right)$$

since $\Delta|v = \Delta(X)|v \cup \Delta(Y) \cup \Delta(X,Y)|v$. The $X$-components and the $Y$-components of the above decomposition are all CNFs. Moreover, when the set $V$ contains a single variable $V$, the above decomposition corresponds to the Shannon decomposition of $\Delta$, which is defined as $\Delta = \left( V \land \Delta|V \right) \lor (\neg V \land \Delta|\neg V)$.
3.4.2 Decomposition by Splitting on Clauses

Another method for constructing \((X, Y)\)-decompositions is by splitting on clauses. That is, each clause \(\gamma\) is split into two sub-clauses \(\alpha\) and \(\beta\), where \(\alpha\) mentions only variables in \(X\) and \(\beta\) mentions only variables in \(Y\). We then take the Cartesian product of these sub-clauses. This is formalized next.

**Definition 3** (Clausal Decomposition [PD10a]). Consider a CNF \(\Delta = \{\gamma_1, \ldots, \gamma_k\}\) defined over disjoint sets of variables \(X\) and \(Y\), where each clause has variables in \(X\) and in \(Y\). Let \(\gamma_i = \alpha_i \lor \beta_i\), where \(\alpha_i\) and \(\beta_i\) are the sub-clauses of \(\gamma_i\) mentioning variables \(X\) and \(Y\), respectively. The clausal \((X, Y)\)-decomposition of CNF \(\Delta\) is defined as

\[
CD(\Delta, X, Y) = \left\{ \left( \bigcup_{i \in S} \alpha_i, \bigcup_{j \in S^c} \beta_j \right) \mid S \subseteq \{1, \ldots, k\} \right\}.
\]

This clausal decomposition allows us to write CNF \(\Delta\) as follows

\[
\Delta = \bigvee_{S \subseteq \{1, \ldots, k\}} \left( \bigwedge_{i \in S} \alpha_i \right) \land \left( \bigwedge_{j \not\in S} \beta_j \right).
\]

More generally, consider a CNF \(\Delta\) defined over disjoint sets of variables \(X\) and \(Y\), and suppose that the CNF is partitioned into \(\Delta(X), \Delta(Y),\) and \(\Delta(X, Y)\) as before. Suppose further that \(\{(L_1, R_1), \ldots, (L_n, R_n)\}\) is the clausal \((X, Y)\)-decomposition of CNF \(\Delta(X, Y)\). The following is then guaranteed to be an \((X, Y)\)-decomposition of CNF \(\Delta\):

\[
\left\{ \left( \Delta(X) \cup L_1, \Delta(Y) \cup R_1 \right), \ldots, \left( \Delta(X) \cup L_n, \Delta(Y) \cup R_n \right) \right\}.
\]

The \(X\)-components of this decomposition have the form \(\Delta(X) \cup L_i\), where \(L_i\) is an \(X\)-component of the clausal decomposition for \(\Delta(X, Y)\). As we shall see later, the number of these components will play a major role in defining our new notion of width.
3.5 More on Vtrees

Before discussing our compilation algorithm, we will introduce some definitions about vtrees that will be used later.

A vtree node $v$ is called a Shannon node iff its left child is a leaf. In this case, the variable labeling the left child is called the Shannon variable of node $v$. In Figure 3.1(a), vtree nodes 1 and 3 are Shannon nodes, with $X$ and $Y$ as their Shannon variables. Accordingly, a right-linear vtree can be seen as one whose every internal node is a Shannon node.

Let $\pi$ be a variable ordering. The right-linear vtree induced by $\pi$ is the one whose in-order traversal visits leaves in the same order as $\pi$. The following is a right-linear vtree induced by order $A, B, C, D$.

![Diagram of a right-linear vtree induced by order $A, B, C, D$.]

We will find it useful to distribute the clauses of a CNF $\Delta$ on a vtree as follows. Each clause of $\Delta$ is assigned to the lowest vtree node that contains the clause variables. Figure 3.2 depicts an example of how clauses are assigned to vtree nodes. We will use $\text{Clauses}(v)$ to denote the clauses assigned to vtree node $v$. We will also use $\text{CNF}(v)$ to denote the clauses assigned to all nodes in the vtree rooted at $v$.

3.6 Compiling CNFs into Structured DNNF

We will now present an algorithm that compiles a CNF into a DNNF that respects a given vtree. Our compilation method is given by Algorithm 1, which takes as input a vtree $v$ and an auxiliary CNF $S$ over the variables of vtree $v$ ($S$ is initially
Theorem 1. The call \( c2s(v, \{\}) \) returns a DNNF that respects vtree \( v \) and that is equivalent to \( CNF(v) \).

Proof. The proof is by induction on vtree nodes. Base case, which happens when \( v \) is a leaf node, is trivially satisfied by Line 3. Now, let \( v \) be an internal node. As an induction hypothesis, assume that for each vtree node \( v' \) below \( v \), \( c2s(v', S') \) computes a DNNF that respects \( v' \) and that is equivalent to \( CNF(v') \cup S' \), where \( S' \) is a CNF over \( Vars(v') \). During the call to \( v \), we either perform variable splitting (Lines 4–12) or clause splitting (Lines 13–19). In both cases, due to the induction hypothesis, recursive calls \( c2s(v', S') \) must compute structured DNNFs of \( CNF(v') \cup S' \). As variable and clause splittings are both sound methods, the algorithm returns a DNNF of \( CNF(v) \cup S \) that respects \( v \). So, the call \( c2s(v, \{\}) \) returns a DNNF that respects vtree \( v \) and that is equivalent to \( CNF(v) \).
Algorithm 1: c2s(v, S)

cache(v, Δ) is a hash table that maps v and Δ into a DNNF.

terminal(Δ) returns the literal or constant equivalent to Δ.

Input: v : a vtree node, S : a CNF over Vars(v).

Output: A DNNF for CNF(v) ∪ S that respects vtree v.

1 if cache(v, S) ≠ nil then return cache(v, S)
2 C ← Clauses(v)
3 if v is a leaf then return terminal(C ∪ S)
4 if v is a Shannon node then
5 X ← Shannon variable of v
6 if {X} and {¬X} assigned to v then α ← ⊥
7 else if {X} assigned to v then
8 α ← X ∧ c2s(vr, (C ∪ S)|X)
9 else if {¬X} assigned to v then
10 α ← ¬X ∧ c2s(vr, (C ∪ S)|¬X)
11 else
12 α ← (X ∧ c2s(vr, (C ∪ S)|X)) ∨ (¬X ∧ c2s(vr, (C ∪ S)|¬X))
13 else
14 X ← variables in the vtree rooted at v
15 Y ← variables in the vtree rooted at vr
16 Partition S into S1(X), S2(Y), and S3(X, Y)
17 α ← ⊥
18 foreach (L, R) ∈ CD(C ∪ S3, X, Y) do
19 α ← α ∨ (c2s(v, S1 ∪ L) ∧ c2s(v, S2 ∪ R))
20 cache(v, S) ← α
21 return α
More generally, a recursive call \(c_2s(v, S)\) will return a DNNF for \(CNF(v) \cup S\) that respects vtree \(v\). Moreover, the algorithm keeps a cache at every vtree node, which is indexed by the auxiliary CNF \(S\).

### 3.7 A New Complexity Parameter for CNFs

In this section, we will introduce CV-width and show that the time and space complexity of Algorithm 1 is exponential only in CV-width. First, we will study a concept that will be quite useful in defining CV-width.

#### 3.7.1 Counting Components

Our new notion of width and the corresponding complexity analysis of our compilation algorithm depend crucially on counting the number of distinct components of clausal decompositions. The following direct definition of these components facilitates this process.

**Definition 4.** Consider a CNF \(\Delta\) and a set of variables \(X\). Let \(\gamma_1, \ldots, \gamma_n\) be the clauses in \(\Delta\) which mention variables inside and outside \(X\), and let \(\alpha_i\) be the sub-clause of \(\gamma_i\) with variables in \(X\). The \(X\)-components of CNF \(\Delta\) are defined as the following CNFs:

\[
CNFs(\Delta, X) = \{\Delta(X) \cup \Gamma \mid \Gamma \subseteq \{\alpha_1, \ldots, \alpha_n\}\}
\]

where \(\Delta(X)\) is the set of clauses of \(\Delta\) that only mention variables \(X\).

As an example, consider the CNF \(\Delta = \{A, B \lor \neg C \lor D, C \lor \neg D\}\) and the set of variables \(X = \{A, B, C\}\). Then, the \(X\)-components of CNF \(\Delta\) are the following four CNFs:

\[
CNFs(\Delta, X) = \{\{A\}, \{A, B \lor \neg C\}, \{A, C\}, \{A, B \lor \neg C, C\}\}.
\]
Suppose that we split on variables \( V \), leading to CNFs \( \Delta|v \): one CNF for each instantiation \( v \) of variables \( V \). Suppose that we further construct a clausal decomposition for each CNF \( \Delta|v \). We will find it quite useful to count the number of distinct components which are obtained from this process.

**Definition 5.** Consider a CNF \( \Delta \) and disjoint sets of variables \( X \) and \( V \). The \( X|V \)-components of CNF \( \Delta \) are defined as the following CNFs:

\[
CNFs(\Delta, X|V) = \bigcup_v CNFs(\Delta|v, X).
\]

As an example, consider the CNF

\[
\Delta = \{ A \lor D \lor V, B \lor \neg C \lor V, B \lor D \lor \neg V, C \lor D \lor \neg V \}.
\]

If \( X = \{ A, B, C \} \) and \( V = \{ V \} \), then we have the following:

\[
\Delta|V = \{ B \lor D, C \lor D \},
\]

\[
CNFs(\Delta|V, X) = \{ \{ \}, \{ B \}, \{ C \}, \{ B, C \} \},
\]

\[
\Delta|\neg V = \{ A \lor D, B \lor \neg C \},
\]

\[
CNFs(\Delta|\neg V, X) = \{ \{ B \lor \neg C \}, \{ A, B \lor \neg C \} \}.
\]

Hence, the \( X|V \)-components of CNF \( \Delta \) are defined as the following six CNFs:

\[
CNFs(\Delta, X|V) = \{ \{ \}, \{ B \}, \{ C \}, \{ B, C \}, \{ B \lor \neg C \}, \{ A, B \lor \neg C \} \}.
\]

These are all the distinct \( X \)-components obtained by first splitting on variables \( V \) then constructing clausal decompositions.

We will use \( \#CNFs(\Delta, X|V) \) to denote the ceiling of \( \log(|CNFs(\Delta, X|V)|) \), where \( \log 0 \) is defined as 0. Thus, in the above example \( \#CNFs(\Delta, X|V) = 3 \).
3.7.2 Clause-Variable Width

We are now ready to introduce the new notion of width, called CV-width. This new width is based on counting the number of distinct components that arise when decomposing a CNF using a series of splits on variables and clauses.

CV-width is defined for a vtree and a corresponding CNF. The CV-width of a CNF is then defined as the smallest CV-width attained by any of its vtrees. To define CV-width for a given vtree, we need to associate a set of clauses and variables with each internal node in the vtree. These sets are defined next.

**Definition 6.** Consider a CNF $\Delta$ and a corresponding vtree. Each internal vtree node $v$ is associated with the following sets:

- **Context Variables:** Shannon variables of $v$’s ancestors.
- **Cutset Clauses:** empty set if $v$ is a Shannon node; otherwise, clauses with variables inside $v^l$ and inside $v^r$.
- **Context Clauses:** clauses with variables inside and outside $v$, and that do not belong to the cutset.

Figure 3.3 depicts a CNF, a corresponding vtree, and the associated context variables, cutset clauses, and context clauses of vtree nodes.

When Algorithm 1 is decomposing a CNF with respect to a vtree node $v$, it would have already split on its context variables. At this point, the CNF can be decomposed by splitting on its cutset and context clauses. One will always split on cutset clauses. However, whether one would need to split on a particular context clause depends on the specific splits adopted at ancestors. This motivates the following definition of width.

**Definition 7 (CV-width).** Consider a CNF and a corresponding vtree. Let $v$ be an internal vtree node with variables $X$, context variables $V$, cutset clauses
Figure 3.3: A vtree, its cutset clauses, context clauses, and context variables, defined for the CNF \( \{ Y \lor \neg Z, Z \lor Q, \neg X \lor Z, X \lor \neg Y \lor Q \} \).
\( \Delta \), and context clauses \( \Gamma \). The width of node \( v \), denoted \( \text{width}(v) \), is \( |\Delta| + \#\text{CNFs}(\Gamma, X|V) \). The CV-width of the vtree is the largest width of any of its internal nodes minus 1. The CV-width of a CNF is the smallest CV-width attained by any of its vtrees.

Consider the CNF \{Y \lor \neg Z, Z \lor Q, \neg X \lor Z, X \lor \neg Y \lor Q\} and the vtree in Figure 3.1(a). The CV-width of this vtree is 2; see Figure 3.3.

### 3.7.3 Complexity Analysis

The following theorem reveals the time and space complexity of our compilation algorithm.

**Theorem 2.** If vtree \( v \) is defined over \( n \) variables and has CV-width \( w \) and \( \text{CNF}(v) \) has size \( m \), then the call \( \text{c2s}(v, \{\}) \) takes time in \( O(nm^3w) \) and returns a DNNF whose size is in \( O(n^3w) \).

To prove Theorem 2, we need the following lemma, whose proof is delegated to the appendix at the end of the chapter.

**Lemma 1.** Let \( v \) be an internal vtree node with variables \( X \), context variables \( V \), cutset clauses \( \Delta \), and context clauses \( \Gamma \). The following hold when Algorithm 1 starts executing a call \( \text{c2s}(v, S) \):

If \( v \) is a Shannon node, then

(a) \( S \in \text{CNFs}(\Gamma, X|V) \).

If \( v \) is not a Shannon node, then

(b) \( C \subseteq \Delta \);

(c) \( S_1 \cup S_2 \in \text{CNFs}(\Gamma, X|V) \);

(d) \( S_3 \subseteq \Sigma \downarrow X \) where \( \Sigma = \Delta \setminus C \).
In the above lemma, we use \( \Sigma \downarrow X \) to denote the CNF which results from replacing every clause in \( \Sigma \) by its sub-clause that mentions variables in \( X \). For example, if \( \Sigma = \{ A \lor \neg B \lor C, \neg A \lor C \lor \neg D \} \) and \( X = \{ A, B \} \), then \( \Sigma \downarrow X = \{ A \lor \neg B, \neg A \} \).

We next prove Theorem 2.

**Proof (Theorem 2).** Let \( v \) be an internal vtree node with variables \( X \), cutset clauses \( \Delta \), context clauses \( \Gamma \), and context variables \( V \). We will next bound the time spent at node \( v \) and the contribution it makes to the DNNF size during all calls made to node \( v \). By adding these time and size bounds for all internal vtree nodes, we can bound the time and space complexity of Algorithm 1.

Assume that \( v \) is a Shannon node. By Lemma 1(a), \( S \in CNFs(\Gamma, X|V) \). Hence, the number of uncached calls to \( v \) is \( \leq 2^{\mid \Delta \mid + \#CNFs(\Gamma, X|V)} \) since \( \Delta = \emptyset \) for a Shannon node. Moreover, each uncached call to \( v \) will construct a decomposition of size at most 2 by doing \( O(2m) \) work (Lines 4–12). The total contribution of a Shannon node to time complexity is then \( O(m2^{\text{width}(v)}) \). Moreover, the total contribution it makes to the DNNF size is \( O(2^{\text{width}(v)}) \).

Assume now that \( v \) is not a Shannon node. The following observations all follow from Lemma 1. First, by Lemma 1(d), if \( |S_3| = i \) and \( |\Sigma| = k \), then \( 0 \leq i \leq k \). Moreover, there are at most \( \binom{k}{i} \) distinct CNFs \( S_3 \) of size \( i \). Second, by Lemma 1(c), there are at most \( 2^{\#CNFs(\Gamma, X|V)} \left( \binom{k}{i} \right) \) uncached calls to node \( v \) for which \( |S_3| = i \). Moreover, each of these calls will construct a clausal decomposition of size \( 2^{|C|+i} \) on Line 19. Hence, the decompositions constructed at Line 19 will have a total size of
\[
\sum_{i=0}^{k} 2^{\#\text{CNFs}(\Gamma, \mathbf{x} | \mathbf{v})} \binom{k}{i} 2^{\lvert C \rvert + i} = 2^{\#\text{CNFs}(\Gamma, \mathbf{x} | \mathbf{v}) + |C|} \sum_{i=0}^{k} \binom{k}{i} 2^i
\]
\[
= 2^{\#\text{CNFs}(\Gamma, \mathbf{x} | \mathbf{v}) + |C|} 3^k
\]
\[
\leq 3^{\#\text{CNFs}(\Gamma, \mathbf{x} | \mathbf{v}) + |C| + k}
\]
\[
= 3^{\#\text{CNFs}(\Gamma, \mathbf{x} | \mathbf{v}) + |\Delta|} \quad \text{by Lemma 1(b)}
\]
\[
= 3^{\text{width}(v)}.
\]

Computing a clausal decomposition is linear in the CNF size. Hence, the total contribution of node \(v\) to time complexity is \(O(m3^{\text{width}(v)})\). Moreover, the total contribution it makes to the DNNF size is \(O(3^{\text{width}(v)})\). As there are \(O(n)\) \(v\)tree nodes, Algorithm 1 has a total time complexity in \(O(nm3^w)\). Moreover, the structured DNNF it constructs has size in \(O(n3^w)\).
3.A Proof of Lemma 1

To prove Lemma 1, we will make use of the following three lemmas.

**Lemma 2.** Consider a CNF and its corresponding vtree. Let $v$ be an internal vtree node with variables $X$ and context variables $V$. Let $\Sigma$ be the clauses in the CNF that mention variables inside and outside $X$. If a call $c2s(v, S)$ is made to node $v$, then $S \in \text{CNFs}(\Sigma, X|V)$.

**Proof.** Note that $S$ is over variables $X$ and it must be obtained from the original clauses of the CNF through a (possibly) series of variable and clause splittings. Thus, clauses in $S$ are sub-clauses of those in $\Sigma$. Consider the path from the root to $v$. To reach $v$, at each node on the path, Algorithm 1 performs either variable or clause splitting. So, there must be a sequence $K$ of variable and clause splittings that would lead us to $S$ when applied on $\Sigma$. Let $v$ be the instantiation of $V$ that presents in $K$. Since we must have performed variable splitting using $v$ on $\Sigma$, clauses in $S$ must be sub-clauses of those in $\Sigma|v$. We know that each clause in $\Sigma$ has variables in $X$ and $\overline{X}$. Let $\Sigma|v = \Delta(X) \cup \Delta(X, \overline{X})$. Here, $\Delta(X)$ must be some of the clauses in $S$, i.e., $\Delta(X) \subseteq S$. Otherwise, $K$ cannot be the sequence that would lead us to $S$. Assume $S = \Delta(X) \cup S'$. So, clauses in $S'$ must be sub-clauses in $\Delta(X, \overline{X})$. Otherwise, again, $K$ cannot be the sequence that would lead us to $S$. More precisely, let $\Delta(X, \overline{X}) = \{\gamma_1, \ldots, \gamma_n\}$ and let $\alpha_i$ be the sub-clause of $\gamma_i$ with variables in $X$. Then, $S'$ is a subset of $\{\alpha_1, \ldots, \alpha_n\}$. Therefore, $S \in \text{CNFs}(\Sigma|v, X)$, which implies $S \in \text{CNFs}(\Sigma, X|V)$.

**Lemma 3.** Let $\Delta = \Delta_1 \cup \Delta_2$ be a CNF, where $\Delta_1$ and $\Delta_2$ are disjoint. Let $X$ be a set of variables. Then, the following holds:

$$\text{CNFs}(\Delta, X) = \{\Sigma_1 \cup \Sigma_2 \mid \Sigma_1 \in \text{CNFs}(\Delta_1, X), \Sigma_2 \in \text{CNFs}(\Delta_2, X)\}.$$
Proof. In the following, for any CNF Σ, we denote the clauses in Σ that only mention variables X by Σ(X). Further, we assume \{δ₁, ..., δₙ\} contains the clauses in ∆ that mention variables inside and outside X. So, we denote by \{α₁, ..., αₙ\} the sub-clauses of δᵢ with variables in X. Finally, we assume \{δ¹₁, ..., δ¹_k\} contains the clauses in ∆₁ that mention variables inside and outside X, and likewise \{δ²₁, ..., δ²_l\} is the clauses in ∆₂ that mention variables inside and outside X. So, we denote by \{α¹₁, ..., α¹_k\} the sub-clauses of δ¹ᵢ with variables in X, and similarly \{α²₁, ..., α²_l\} denotes the sub-clauses of δ²ᵢ with variables in X. Note that

\[
\{δ₁, ..., δₙ\} = \{δ¹₁, ..., δ¹_k\} \cup \{δ²₁, ..., δ²_l\},
\]

\[
\{α₁, ..., αₙ\} = \{α¹₁, ..., α¹_k\} \cup \{α²₁, ..., α²_l\}.
\]

Then, we have the following equations:

\[
CNFs(∆, X) = \{∆(X) \cup Γ | Γ \subseteq \{α₁, ..., αₙ\}\}
\]

\[
= \{∆₁(X) \cup (∆₂(X) \cup Γ) | Γ \subseteq \{α₁, ..., αₙ\}\}
\]

\[
= \{∆₁(X) \cup Γ₁ \cup (∆₂(X) \cup Γ₂) | \Gamma₁ \subseteq \{α¹₁, ..., α¹_k\}, Γ₂ \subseteq \{α²₁, ..., α²_l\}\}
\]

\[
= \{Σ₁ \cup Σ₂ | Σ₁ \in CNFs(∆₁, X), \Sigma₂ \in CNFs(∆₂, X)\}.
\]

Lemma 4. Let v be an internal vtree node, which is not Shannon, with variables X, cutset clauses ∆, context clauses Γ and context variables V. If a call c₂s(v, S) is made to node v by Algorithm 1, then S₁ ∪ S₂ ∈ CNFs(Γ, X|V), and S₃ ∈ CNFs(∆ \ C, X|V).

Proof. By Lemma 2, S ∈ CNFs(Σ, X|V) with Σ being clauses that mention variables inside and outside X. That means, there exists an instantiation v of V such that S ∈ CNFs(Σ|v, X). Note that, for a vtree node that is not Shannon,
\[ \Sigma = \Gamma \cup (\Delta \setminus C), \text{ where } \Gamma \text{ and } \Delta \setminus C \text{ are disjoint.} \]

Then, we have the following:

\[ S \in \text{CNFs}(\Gamma|v \cup (\Delta \setminus C)|v, X) \]

\[ S \in \{ \Sigma_1 \cup \Sigma_2 \mid \Sigma_1 \in \text{CNFs}(\Gamma|v, X), \Sigma_2 \in \text{CNFs}((\Delta \setminus C)|v, X) \} \]

\[ S_3 \in \text{CNFs}((\Delta \setminus C)|v, X) \]  
\[ S_3 \in \text{CNFs}(\Delta \setminus C, X|V) \]

\[ S_1 \cup S_2 \in \text{CNFs}(\Gamma|v, X) \]  
\[ S_1 \cup S_2 \in \text{CNFs}(\Gamma, X|V). \]  

Equation (3.1) is due to Lemma 3. Equation (3.2) is due to \( S_3 \) consists of clauses that mention both variables in \( v^l \) and \( v^r \), but none of the clauses in \( \text{CNFs}(\Gamma|v, X) \) can mention both variables in \( v^l \) and \( v^r \). Equation (3.3) follows from Definition 4. Equation (3.4) is due to \( S_1 \cup S_2 \) does not contain any clause that mentions both variables in \( v^l \) and \( v^r \), but each clause in \( \text{CNFs}((\Delta \setminus C)|v, X) \) must mention both variables in \( v^l \) and \( v^r \). Equation (3.5) follows from Definition 4.

We are now ready to prove Lemma 1.

**Proof** (Lemma 1). Item (a) is an immediate corollary of Lemma 2, as for a Shannon node clauses that mention variables inside and outside \( X \) are its context clauses \( \Gamma \). Item (b) holds because of the way clauses are distributed over vtree nodes. Item (c) is due to Lemma 4. For Item (d), by Lemma 4, we know that \( S_3 \in \text{CNFs}(\Sigma, X|V) \). This simply implies that \( S_3 \) is obtained from some of the clauses in \( \Sigma \) by replacing them with their sub-clauses that mention variables \( X \). That is, \( S_3 \subseteq \Sigma \downarrow X \).
CHAPTER 4

On Compiling CNF into Deterministic Subsets of DNNF

In Chapter 3, we have introduced CV-width to parameterize the compilation of CNFs into DNNFs. In this chapter, we will target deterministic subsets of DNNFs, which are crucial for performing probabilistic reasoning. In particular, we will introduce decision-width and linear CV-width to parameterize the compilation of CNFs into Decision-DNNFs, SDDs, and OBDDs. We will show that compiling CNFs into both Decision-DNNFs and SDDs can be done in time and space that are exponential only in decision-width. We will also show CNFs can be compiled into OBDDs in time and space that are exponential only in linear CV-width. These complexity results complement the results shown in Chapter 3 as linear CV-width is a special case of decision-width which is in turn a special case of CV-width. The material in this chapter is based on the work published in [OD14a, OD14b].

4.1 Introduction

Model counting is the problem of determining the number of satisfying assignments of a propositional sentence. Being a \#P-complete problem [Val79], model counting is central to many AI problems such as probabilistic reasoning [Rot96, Dar02], and state-of-the-art model counters have been successfully used for doing probabilistic inference [CD05, SBK05, CDJ06, CD08]. Up-to-date, those model counters have been based on two approaches that have proven effective in practice.
One approach is based on DPLL [DP60, DLL62], which is a family of algorithms that were initially developed for SAT: the problem of deciding whether a propositional sentence has a satisfying assignment. In essence, it is a systematic search algorithm that searches the space of truth assignments until finding a satisfying one or identifying that such an assignment does not exist. This search method can easily be extended to compute the number of satisfying assignments of the sentence. Simply, by not stopping the search when a single satisfying assignment is found, and exhaustively continuing to look for all other satisfying assignments, one can obtain a naive model counting algorithm, called exhaustive DPLL [BL99]. To make this approach more effective, various sophisticated techniques were incorporated into the core exhaustive DPLL algorithm, such as component analysis [BP00] and component caching [ML98]. The latter technique is used to avoid counting the models of the same components multiple times. The former technique is used to identify disconnected components and count their models independently to improve efficiency.

Another approach for model counting is based on knowledge compilation, where the idea is to construct a propositional representation that supports model counting in polynomial time. As we discussed earlier, two fundamental properties on NNF sentences ensure the tractability of model counting: decomposability and determinism. Although these two properties characterize d-DNNF, which is the most general language known that supports efficient model counting, a strict subset, called Decision-DNNF [HD07], has been used in state-of-the-art model counters based on knowledge compilers [Dar04, MMB12].

Although the approaches outlined above look conceptually different than each other, a strong connection between them has been established [HD07]. In particular, the traces\(^\text{1}\) of the searches performed by state-of-the-art model counters has been shown to be in Decision-DNNF. In other words, model counters based on

---

\(^{1}\)See Section 4.3 for a detailed discussion on the trace of an exhaustive search algorithm.
exhaustive DPLL effectively generates the compilation of the Boolean sentence in Decision-DNNF. By this result, Decision-DNNF has the role of bridging model counters and knowledge compilers. More importantly, any new result pertaining to Decision-DNNFs will have a possibly significant further impact on model counters. For instance, the relationship between Decision-DNNFs and FBDDs has been recently studied in [BLR13]. Accordingly, Decision-DNNFs can be converted into FBDDs with only a quasipolynomial increase in the representation size.\textsuperscript{2} This result allowed the authors to show new exponential lower bounds on Decision-DNNFs, by leveraging the existing lower bounds on FBDDs, which are immediately applicable to model counters.

In this chapter, we present new results on compiling CNFs into deterministic subsets of DNNF, which are crucial for performing model counting. First, we introduce a new notion of width for CNFs, called decision-width. This new width is a special case of CV-width, and is characterized by a special kind of vtree, called decision vtree. We show a compilation algorithm that can compile CNFs into Decision-DNNFs in time and space that are exponential only in decision-width. As will be shown later in Chapter 5, this result not only improves the existing complexity results on d-DNNF compilation but also the existing results on the complexity of model counting. Second, we show that Decision-DNNFs constructed by our algorithm can be converted to SDDs in linear time, which will lead to a tighter bound on the complexity of compiling CNFs into SDDs. Third, we introduce linear CV-width, which is a special case of decision-width (characterized by right-linear vtrees), and show that CNFs can be compiled into OBDDs in time and space that are exponential only in linear CV-width. This result will also lead to a tighter bound on the complexity of CNF-to-OBDD compilation.

This chapter is structured as follows. We start by introducing decision-width and discussing a compilation algorithm that can compile CNFs into Decision-

\textsuperscript{2}A quasipolynomial grows slightly faster than a polynomial, but not exponentially fast.
DNNFs in time and space that are exponential only in this width (Section 4.2). We next discuss the importance of Decision-DNNFs in model counting by reviewing in detail the strong connection that has been established between Decision-DNNFs and model counters (Section 4.3). We then show that the output of our algorithm for compiling CNFs into Decision-DNNFs can be transformed in linear time to SDDs (Section 4.4). We finally introduce linear CV-width and show a CNF-to-OBDD compilation algorithm parameterized by this width (Section 4.5).

4.2 Compiling CNFs into Decision-DNNFs

The purpose of this section is to show an algorithm that compiles CNFs into Decision-DNNFs with a complexity guarantee. To analyze the complexity of the algorithm, we will also introduce a new notion of width and study its properties.

4.2.1 Decision-DNNF

A decision node is a special form of a disjunction node which is depicted as follows:

\[ \bigvee \quad \bigwedge \]

\[ X \quad \alpha \quad \neg X \quad \beta \]

where \( X \) is a variable and \( \alpha \) and \( \beta \) are arbitrary NNF nodes. A d-DNNF is called a Decision-DNNF iff each of its disjunction nodes is a decision node; see Figure 4.1(b). In this case, determinism is always ensured by the decision nodes. Moreover, both FBDD and OBDD are a strict subset of Decision-DNNF.
4.2.2 Decision Vtrees

Both the width and the compilation algorithm we will present in this section are driven by a special type of vtree, which is introduced next.

**Definition 8** (decision vtree). A clause is compatible with an internal vtree node $v$ iff the clause mentions some variables inside $v^l$ and some variables inside $v^r$. A vtree for CNF $\Delta$ is said to be a decision vtree for $\Delta$ iff every clause in $\Delta$ is compatible with only Shannon nodes.\(^3\)

Figure 4.1(a) depicts a decision vtree for the CNF \{Y ∨ ¬Z, ¬X ∨ Z, X ∨ ¬Y, X ∨ Q\}. Indeed, one can always construct a decision vtree for any CNF: the simplest way to do this is to construct a right-linear vtree, which is guaranteed to be a decision vtree. Moreover, due to the compatibility property, cutset clauses of vtree nodes will always be empty in a decision vtree.

\(^3\)A unit clause (one containing a single literal) is not compatible with any vtree node. Hence, a unit clause trivially satisfies the condition of being compatible with only Shannon nodes.
4.2.3 A Compilation Algorithm

We will next present an algorithm that compiles a CNF into a Decision-DNNF using a decision vtree for the CNF. This compilation method is given by Algorithm 2, which takes as input a decision vtree $v$ and an auxiliary CNF $S$ over the variables of vtree $v$ ($S$ is initially empty). The CNF $\Delta$ to be compiled is passed with the vtree as follows. Each clause of $\Delta$ is assigned to the lowest vtree node that contains the clause variables. Figure 4.2 depicts an example of how clauses are assigned to vtree nodes.\footnote{As before, we will use $\text{Clauses}(v)$ to denote the clauses assigned to a vtree node $v$ and use $\text{CNF}(v)$ to denote the clauses assigned to all nodes in the vtree rooted at $v$.} Note that the (non-unit) clauses are assigned only to Shannon nodes as the vtree is a decision vtree. A recursive call $\text{c2d}(v, S)$ will return a Decision-DNNF for $\text{CNF}(v) \cup S$. The algorithm keeps a cache at every vtree node, which is indexed by $S$.

Indeed, Algorithm 2 is a special case of Algorithm 1 presented in Chapter 3. The difference is that we require a decision vtree as input, instead of a general vtree. This simplifies Algorithm 2 while processing a non-Shannon vtree node (Lines 10–13). Here, the CNF would already be decomposed with respect to the current vtree node (due to the compatibility property of decision vtrees). Hence,
Algorithm 2: c2d(v, S)

`cache(v, ∆)` is a hash table that maps `v` and `∆` into a Decision-DNNF.

`terminal(∆)` returns the literal or constant equivalent to `∆`.

**Input:** `v`: a vtree node, `S`: a CNF over `Vars(v)`.

**Output:** A Decision-DNNF for `CNF(v) ∪ S`.

1. if `cache(v, S) ≠ nil` then return `cache(v, S)`
2. `C ← Clauses(v)`
3. if `v` is a leaf then return `terminal(C ∪ S)`
4. if `v` is a Shannon node then
   5. `X ←` variable of `v^l`
   6. if `{X}` and `{-X}` assigned to `v^l` then `α ← ⊥`
   7. else if `{X}` assigned to `v^l` then `α ← X ∧ c2d(v^r, (C ∪ S){X})`
   8. else if `{-X}` assigned to `v^l` then `α ← ¬X ∧ c2d(v^r, (C ∪ S){¬X})`
   9. else `α ← (X ∧ c2d(v^l, (C ∪ S){X}) ∨ (¬X ∧ c2d(v^l, (C ∪ S){¬X}))`
10. else
11. `S_1 ←` clauses in `S` that only mention variables in `v^l`
12. `S_2 ←` clauses in `S` that only mention variables in `v^r`
13. `α ← (c2d(v^l, S_1) ∧ c2d(v^r, S_2))`
14. `cache(v, S) ← α`
15. return `α`

we do not need to construct a clausal decomposition as done by Algorithm 1 in Chapter 3 (Lines 13–19). Because of this and the fact that each disjunction node created by the algorithm is a decision node (Line 9), Algorithm 2 constructs a Decision-DNNF, as opposed to a DNNF. In the following, we prove the soundness of the algorithm.

**Lemma 5.** Let `v` be a decision vtree for `CNF(v)`. Let `S` be a CNF over `Vars(v)` whose clauses are compatible with only Shannon nodes of `v`. The call `c2d(v, S)` to Algorithm 2 returns a Decision-DNNF equivalent to `CNF(v) ∪ S`. 

54
Proof. The proof is by induction on vtree nodes. The base case is when \( v \) is a leaf node. This case is trivially satisfied by Line 3. Assume now that \( v \) is an internal node. As an induction hypothesis, consider that for each vtree node \( v' \) below \( v \), the call \( c2d(v', S') \) computes a Decision-DNNF equivalent to \( CNF(v') \cup S' \), where \( S' \) is a CNF over \( Vars(v') \) whose clauses are compatible with only Shannon nodes of \( v' \).

During the call to \( v \), we will compute a Decision-DNNF equivalent to \( CNF(v) \cup S \) by utilizing the Shannon decomposition of a Boolean function: \( f \equiv (X \land f_X) \lor (\neg X \land f_{\neg X}) \). Note that in our context \( f = CNF(v) \cup S \). Assume \( v \) is a Shannon node. Then \( v_l \) is a leaf node (labeled with variable \( X \)). The possible clauses that can be assigned to \( v_l \) are \( \{X\} \) and \( \{\neg X\} \). Lines 6–9 consider all four possible assignments of those two clauses to \( v_l \): (1) both \( \{X\} \) and \( \{\neg X\} \) are assigned and \( f \equiv \bot \); (2) only \( \{X\} \) is assigned and \( f \equiv x \land f_x \); (3) only \( \{\neg X\} \) is assigned and \( f \equiv \neg x \land f_{\neg x} \); and (4) no clause is assigned and \( f \equiv (x \land f_x) \lor (\neg x \land f_{\neg x}) \).

Except for the first case, in which \( f \) is trivially computed as \( \bot \), by the induction hypothesis, \( c2d(v^r, (C \cup S)|X) \) and \( c2d(v^r, (C \cup S)|\neg X) \) compute Decision-DNNFs for \( f|X \) and \( f|\neg X \), respectively.\(^5\) Note that we construct a disjunction node only in the last case, which is a decision node. So, when \( v \) is a Shannon node, we compute a Decision-DNNF equivalent to \( CNF(v) \cup S \). Assume now that \( v \) is a non-Shannon node. In this case, \( C \) must be empty because the vtree is a decision vtree. Thus, \( CNF(v) \equiv CNF(v^l) \cup CNF(v^r) \). Also, \( S \) cannot contain any clause that mentions variables from both \( v^l \) and \( v^r \) as no clause in \( S \) can be compatible with \( v \). Then, by the induction hypothesis, on Line 13, we compute a Decision-DNNF equivalent to \( CNF(v) \cup S \).

\[\Box\]

Corollary 1. Let \( v \) be a decision vtree for \( CNF(v) \). The call \( c2d(v, \{\} ) \) to Algorithm 2 returns a Decision-DNNF that is equivalent to \( CNF(v) \).

For instance, when the vtree in Figure 4.2 is passed to Algorithm 2, it computes...\(^5\) If a clause is not compatible with a vtree node \( v \) then every conditioning of the clause will also not be compatible with \( v \).
the Decision-DNNF in Figure 4.1(b). To analyze time and space complexities of the algorithm, we next introduce a new notion of width.

4.2.4 Decision-Width

Before defining the new notion of width, we will introduce the following concept.

**Definition 9.** Consider a CNF $\Gamma$ and a set of variables $V$. We denote by $CCNFs(\Gamma, V)$ the set of CNFs that is obtained from conditioning $\Gamma$ on each instantiation $v$ of $V$.

For instance, consider the CNF $\Gamma = \{X \lor Y \lor Z, X \lor Y \lor Q, \neg X \lor \neg Y \lor Z, X \lor \neg Y \lor Z\}$ and the set of variables $V = \{Y\}$. Then,

$$\Gamma|Y = \{\neg X \lor Z, X \lor Z\},$$

$$\Gamma|\neg Y = \{X \lor Z, X \lor Q\},$$

$$CCNFs(\Gamma, V) = \{\{\neg X \lor Z, X \lor Z\}, \{X \lor Z, X \lor Q\}\}.$$  

We are now ready to introduce the new notion of width.

**Definition 10 (Decision-Width).** Consider a CNF $\Delta$ and a corresponding decision vtree. Let $v$ be an internal vtree node with context clauses $\Gamma$. Let $Y$ be variables outside $v$. Then, the width of $v$ is the ceiling of $\log(|CCNFs(\Gamma, Y)|)$, where $\log 0$ is defined as 0. The decision-width of the decision vtree is the largest width of any of its internal nodes minus 1. The decision-width of the CNF is the smallest decision-width attained by any of its decision vtrees.

For instance, consider the vtree in Figure 4.1(a). Assuming that the vtree corresponds to the CNF $\{Y \lor \neg Z, \neg X \lor Z, X \lor \neg Y, X \lor Q\}$, the width of node $v = 2$ is 1, and the decision-width of the vtree is 0.

We note that decision-width is a special case of CV-width. In particular, the decision-width of a decision vtree will always be the same as the CV-width of
the vtree. This is because the width defined in Definition 7 reduces to the width defined above when applied to a decision vtree. That is, for an internal vtree node \( v \) with variables \( \mathbf{X} \), outside variables \( \mathbf{Y} \), context variables \( \mathbf{V} \), cutset clauses \( \Delta \), and context clauses \( \Gamma \), we have
\[
\log(|\text{CCNFs}(\Gamma, \mathbf{X}|\mathbf{V})|) = |\Delta| + \#\text{CNFs}(\Gamma, \mathbf{X}|\mathbf{V}).
\]
Here, \( \Delta \) will be empty due to the compatibility property of decision vtrees. Also, in general, \( \Gamma \) will be defined over variables \( \mathbf{X} \) and \( \mathbf{Y} \). Yet, due to the compatibility property of decision vtrees, \( \Gamma \) will be defined over variables \( \mathbf{X} \) and \( \mathbf{V} \) (i.e., the variables of \( \mathbf{Y} \) that could appear in \( \Gamma \) can only come from context variables \( \mathbf{V} \)). Therefore, we have
\[
\#\text{CNFs}(\Gamma, \mathbf{X}|\mathbf{V}) = \log(|\text{CCNFs}(\Gamma, \mathbf{Y})|),
\]
which leads to the following result.

**Proposition 1.** The decision-width of a decision vtree is the same as the CV-width of the vtree.

Having defined decision-width, we can now establish the complexity of Algorithm 2.

**Theorem 3.** If decision vtree \( v \) is over \( n \) variables and has decision-width \( w \), and if \( \text{CNF}(v) \) has size \( m \), then the call \( \text{c2d}(v, \{\}) \) to Algorithm 2 takes time in \( O(nm^2w) \) and returns a Decision-DNNF whose size is in \( O(n2^w) \).

**Proof.** Each distinct call to a Shannon node (Lines 4–9) takes time in \( O(2m) \): we perform at most two conditionings of \( C \cup S \), which has at most \( m \) clauses, on a single literal. This process contributes to the size at most three nodes, each having two children. Each distinct call to a non-Shannon node (Lines 10–13) takes time in \( O(m) \): we partition the set \( S \), which has at most \( m \) clauses, into two subsets. This case contributes to the size one node with two children. Also, due to caching, the number of distinct calls to a vtree node \( v \) is at most \( 2^k \) where \( k \) is the width of \( v \) (see the proof of Theorem 2). As there are \( O(n) \) nodes in the vtree, Algorithm 2 takes time in \( O(nm^2w) \) and returns a Decision-DNNF whose size is in \( O(n2^w) \).
4.3 Decision-DNNFs and Model Counters

In this section, we will discuss the close relationship between Decision-DNNFs and model counters based on exhaustive DPLL.

Exhaustive DPLL counts the models of a propositional sentence by searching the space of truth assignments until identifying all the satisfying ones. In particular, given a propositional sentence \( \Delta \), it chooses a variable \( X \) of \( \Delta \), and then considers two cases recursively, which correspond to \( \Delta|X \) and \( \Delta|\neg X \). It then obtains the model count of \( \Delta \) by adding up the model counts of \( \Delta|X \) and \( \Delta|\neg X \). In fact, exhaustive DPLL can be seen as constructing a tree. For example, the tree in Figure 4.3(a) shows all the paths that are traversed during an exhaustive DPLL on a propositional sentence. Each circled node represents a variable on which two decisions are performed: the variable is either set to false (dashed edge) or set to true (solid edge). This way, paths from the root to leaf nodes represent (partial) variable assignments. Each leaf node then represents the result of the search when the variable assignment on the path from the root to the leaf is applied on the sentence, with the label \text{unsat} being unsatisfiable and the label \text{sat} being satisfiable.
**sat** being satisfiable. This tree is called the *trace* of the search performed by an exhaustive DPLL [HD07]. Here, one can think of each circled node of the tree as a disjunction node, by utilizing the following:

\[
\begin{align*}
&X \\
&\quad \alpha \\
&\quad \beta
\end{align*}
\]

\[
\begin{align*}
&\land \quad \lor \\
&\quad X \\
&\quad \alpha \\
&\quad \neg X \\
&\quad \beta
\end{align*}
\]

Figure 4.3(b) shows the tree obtained from Figure 4.3(a) using the above conversion, and also replacing each **sat** with **⊤**, and each **unsat** with **⊥**. From this structure, by replacing the same nodes with unique nodes, one would obtain an equivalent NNF, which turns out to satisfy both decomposability and determinism. In fact, the traces of exhaustive DPLL correspond to FBDDs [HD07]. Moreover, when exhaustive DPLL is augmented with component analysis, its traces correspond to Decision-DNNFs [HD07].

This close connection between exhaustive DPLL and Decision-DNNFs has two implications. First, it allows one to translate lower bounds on Decision-DNNFs immediately into lower bounds on the complexity of model counters. Second, but under some assumptions, it allows one to translate Decision-DNNF upper bounds into ones on the complexity of model counters. For example lower bounds, it was recently shown that Decision-DNNFs can be converted into FBDDs with only a quasipolynomial increase in size [BLR13]. As a result, known lower bounds for FBDDs immediately translate into lower bounds on model counters whose traces are in Decision-DNNF (see [BLR13] for examples).

Translating upper bounds on Decision-DNNF to upper bounds on arbitrary model counters, however, is not as direct. Here, one needs, for example, to assume that the traces of the model counter are optimal, and that the time complexity of the counter is polynomial in the size of the trace. Under these assumptions, an upper bound on Decision-DNNF translates directly into an upper bound on the
model counter. Interestingly enough, Algorithm 2 satisfies the second condition. The algorithm does not satisfy the first condition, but we know that its traces (i.e., compilations) are bounded exponentially only by the decision-width. Since this width dominates the primal graph treewidth (see Chapter 5), we now have a tighter upper bound on model counting in general (realized by Algorithm 2). We also have a tighter upper bound on any model counter that satisfies the previous conditions.

4.4 From Decision-DNNF to SDD

We next show a new complexity result on compiling CNFs into SDDs. In particular, we show an upper bound that is exponential only in decision-width. We will obtain this result by showing that Decision-DNNFs generated by Algorithm 2 can be converted into compressed and trimmed SDDs in linear time and by at most doubling the size. That is, Algorithm 2 is effectively compiling SDDs.

Note that the output of Algorithm 2 is a special form of Decision-DNNF. In particular, the vtree used in the compilation provides the generated Decision-DNNF with a specific structure. That is, every node $N$ in the Decision-DNNF is associated with some vtree node $v$ in the following way:

- a conjunction node is associated with $v$ when $\text{Vars}(N^l) \subseteq \text{Vars}(v^l)$ and $\text{Vars}(N^r) \subseteq \text{Vars}(v^r)$; and

- a disjunction node, $(X \land \alpha) \lor (\neg X \land \beta)$, is associated with $v$ when $X \subseteq \text{Vars}(v^l)$ and $\text{Vars}(\alpha \cup \beta) \subseteq \text{Vars}(v^r)$.

For instance, each node of the Decision-DNNF in Figure 4.1(b) is associated with a vtree node in Figure 4.1(a).

Algorithm 3 shows how to convert a Decision-DNNF into an SDD. It takes a Decision-DNNF that is constructed by Algorithm 2 and computes two compressed
Algorithm 3: d2sdd(N)

`cache(N)` is a hash table that maps `N` to an SDD. 

`terminal(N)` returns the terminal SDD equivalent to `F(N)`. 

`unique(α)` removes an element from `α` if its prime is `⊥`. It then returns `s` if `α = {(p₁, s), (p₂, s)}` or `α = {(⊤, s)}`; returns `p₁` if `α = {(p₁, ⊤), (p₂, ⊥)}`; else returns the unique SDD node with elements `α`.

**Input:** `N`: a Decision-DNNF generated by Algorithm 2.

**Output:** Two compressed and trimmed SDDs equivalent to `F(N)` and `¬F(N)`.

1. if `cache(N) ≠ nil` then return `cache(N)`
2. if `N` is a leaf node then return `terminal(N), terminal(¬N)`
3. if `N = (X ∧ N₁) ∨ (¬X ∧ N₂)` then
   4. `s₁, ¬s₁ ← d2sdd(N₁)`
   5. `s₂, ¬s₂ ← d2sdd(N₂)`
   6. `α ← unique({(X, s₁), (¬X, s₂)}`
   7. `¬α ← unique({(X, ¬s₁), (¬X, ¬s₂)}`
4. else // `N = N_l ∧ N_r`
   9. `p, ¬p ← d2sdd(N_l)`
   10. `s, ¬s ← d2sdd(N_r)`
   11. `α ← unique({(p, s), (¬p, ⊥)}`
   12. `¬α ← unique({(p, ¬s), (¬p, ⊤)}`
13. `cache(N) ← α, ¬α`
14. return `α, ¬α`

and trimmed SDDs that are equivalent to the Boolean functions represented by `N` and the negation of `N`. The conversion is done in a bottom-up fashion. Terminal SDDs are obtained from leaf nodes (Line 2). Then, a disjunction node (Lines 3–7) or a conjunction node (Lines 8–12) is constructed using the results of recursive calls. To prevent redundant calculations, the results are cached (Line 13). The following theorem establishes the soundness and the complexity of the algorithm.
Theorem 4. If $N$ is a Decision-DNNF generated by Algorithm 2 and has size $m$, then the call $\text{d2sdd}(N)$ takes time in $O(m)$, and returns two compressed and trimmed SDDs for $F(N)$ and $\neg F(N)$, whose sizes are in $O(m)$.

Proof. The proof is by induction on NNF nodes. The base case is when $N$ is a leaf node, which is satisfied by Line 2. As an induction hypothesis (IH), assume that for every NNF $N'$ below $N$ whose size is $k$, the call $\text{d2sdd}(N')$ takes time in $O(k)$, and returns two compressed and trimmed SDDs for $F(N')$ and $\neg F(N')$, whose sizes are in $O(k)$. Suppose that $N$ is a disjunction node, of the form $(X \land N_1) \lor (\neg X \land N_2)$, where $N_1$ and $N_2$ are NNF nodes. Let $|N_1| = m_1$ and $|N_2| = m_2$. Then, $m = 6 + m_1 + m_2$. By the IH, the call $\text{d2sdd}(N_1)$ (Line 4) takes time in $O(m_1)$, and returns the SDDs for $F(N_1)$ and $\neg F(N_1)$, whose sizes are in $O(m_1)$. Similarly, the call $\text{d2sdd}(N_2)$ (Line 5) takes time in $O(m_2)$, and returns the SDDs for $F(N_2)$ and $\neg F(N_2)$, whose sizes are in $O(m_2)$. Then, using the structure of $N$, we construct the SDD for $F(N)$ (Line 6), whose size is at most $2 + O(m_1) + O(m_2) = O(m)$. To construct the SDD for $\neg F(N)$, we just need to negate the subs of the SDD for $F(N)$ (Line 7), which are already constructed by the recursive calls. So, the call to a disjunction node takes time in $O(m)$. Assume now that $N$ is a conjunction node, of the form $N^l \land N^r$, where $N^l$ and $N^r$ are NNF nodes. Let $|N^l| = m_1$ and $|N^r| = m_2$. Then, $m = 2 + m_1 + m_2$. By the IH, the call $\text{d2sdd}(N^l)$ (Line 9) takes time in $O(m_1)$, and returns the SDDs for $F(N^l)$ and $\neg F(N^l)$, whose sizes are in $O(m_1)$. Similarly, the call $\text{d2sdd}(N^r)$ (Line 10) takes time in $O(m_2)$, and returns the SDDs for $F(N^r)$ and $\neg F(N^r)$, whose sizes are in $O(m_2)$. We then construct the SDD for $F(N)$ (Line 11) by making use of the following: $F(N) = (F(N^l) \land F(N^r)) \lor (\neg F(N^l) \land \bot)$. Thus, the constructed SDD has size at most $2 + O(2m_1) + O(m_2) = O(2m)$. Again, the SDD for $\neg F(N)$ is constructed by negating the subs (Line 12). Thus, the call to a conjunction node takes time in $O(m)$. ■

For instance, when we pass the Decision-DNNF in Figure 4.1(b) to Algorithm 3,
Figure 4.4: An SDD and its negation (both respect the vtree in Figure 4.1(a)).

4.5 Compiling CNFs into OBDDs

We can immediately use Algorithm 2 to construct OBDDs: all we need is to ensure that the input vtree is right-linear. In this case, since every internal vtree node is a Shannon node (and hence the vtree is a decision vtree), Lines 4–9 will always be invoked to construct a Shannon decomposition. This essentially creates an OBDD which respects the variable order underlying the right-linear vtree. To state the complexity of Algorithm 2, we will use a restricted version of decision-width, which is defined for right-linear vtrees.

**Definition 11.** The linear CV-width of a CNF is the smallest decision-width attained by any right-linear vtree of the CNF.

Note that linear CV-width is a special case of decision-width, which is a special case of CV-width. By Theorem 3, we know that if a CNF has $n$ variables and has a linear CV-width $w$, it must have an OBDD whose size is in $O(n^{2^w})$. 
CHAPTER 5

A Comparison of New and Classical CNF Parameters

We have so far introduced new complexity parameters for CNFs, together with accompanying compilation algorithms that are bounded by such parameters. In this chapter, we will provide a comparison of those new parameters with some classical parameters that characterize structural properties of CNFs, leading to new complexity results in knowledge compilation. In particular, we consider three parameters: treewidth, cutwidth, and pathwidth. The first parameter is a property of some graph abstraction of the CNF, such as primal, dual, and incidence graphs, and has been used to bound the size of DNNF compilations. The last two parameters apply directly to a CNF and have been used to bound the size of OBDD compilations. We will show that CV-width strictly dominates the treewidth of the incidence CNF graph, decision-width strictly dominates the treewidth of the primal CNF graph, and linear CV-width strictly dominates the cutwidth and pathwidth of a CNF. The material in this chapter is based on the work published in [OD14a, OD14b].

This chapter is structured as follows. We start with defining treewidth, cutwidth, pathwidth, and other necessary notions needed to provide a comparison with new complexity parameters (Section 5.1). We then present our comparison results (Section 5.2). This is followed by a discussion on the results of the comparison, where we provide new complexity results for compiling DNNF and its subsets (Section 5.3). Proofs of the technical results appear in the appendix at the end.
5.1 Classical CNF Parameters

In this section, we will study some classical CNF parameters that have been commonly used in knowledge compilation, namely, treewidth, cutwidth, and pathwidth. Before defining these parameters, we will review the notion of dominance that will be employed later to make a comparison among complexity parameters. As is customary in fixed-parameter tractability [SS09], we will use the following definition.

Definition 12. Let $p_1$ and $p_2$ be two parameters defined for CNFs. We say that $p_1$ dominates $p_2$ iff there exists a computable function $f$ such that for every CNF $\Delta$ we have

$$p_1(\Delta) \leq f(p_2(\Delta)).$$

Moreover, $p_1$ strictly dominates $p_2$ iff $p_1$ dominates $p_2$ but not vice versa. Finally, $p_1$ and $p_2$ are incomparable iff neither dominates the other.

Accordingly, strict domination of $p_1$ over $p_2$ implies that $p_1$ is effectively no greater and can be bounded when $p_2$ is unbounded.\(^1\) We next briefly review treewidth.

5.1.1 Treewidth

Treewidth is a well-known graph-theoretic property that measures how close a graph to a tree. It is defined over undirected graphs and is a number between 0 and $n - 1$ where $n$ is the number of vertices of the graph. Graphs with tree structure have treewidth of 1 (assuming at least an edge exists in the tree), whereas complete graphs have treewidth of $n - 1$. Treewidth is a fundamental property for parameterized complexity as many hard graph problems arising from various areas (e.g., probabilistic reasoning and constraint satisfaction) become easy on graphs with bounded treewidth. Other than its original definition in [RS84], there are

---

\(^1\)Unbounded means that the parameter grows with the problem size (e.g., in $n$ where $n$ is the number of CNF variables).
several, equivalent definitions of treewidth (e.g., [Bod05]). We will make use of the following definition based on eliminating vertices of a graph.

An elimination order of an undirected graph is an ordering of the vertices of the graph. Suppose that we are given an elimination order \( \pi = V_1, \ldots, V_n \) for an undirected graph \( G \) of \( n \) vertices. We will eliminate vertices in graph \( G \), in the order \( \pi \) (i.e., first \( V_1 \), then \( V_2 \), and so on). To eliminate a vertex is to first connect vertex’s neighbors pairwise and then remove the vertex from the graph. During this process, we will get a number of graphs. Let \( G_i \) be the graph before eliminating node \( V_i \) (so, \( G_1 = G \)). Consider the number of neighbors of \( V_i \) in \( G_i \), for \( 1 \leq i \leq n \). Let \( k \) be the maximum number of such neighbors. Then, \( k \) is the width of \( \pi \). The treewidth of graph \( G \) is the minimum width attained by any elimination order of \( G \).

For instance, Figure 5.1 performs the above procedure on a given undirected graph. Accordingly, the width of the given elimination order \( X, Y, W, Z, Q \) is 3. Indeed, we can find a better elimination order (e.g., \( W, Z, X, Y, Q \)) that has width 2.
Moreover, given a tree with at least one edge, one can always construct an elimination order whose width is 1: just use a topological order of vertices. Hence, the treewidth of such trees is always 1. On the other hand, given a complete graph (a clique more generally), any elimination order will have width of \( n - 1 \) where \( n \) is the number of vertices, as any vertex will have \( n - 1 \) neighbours in the initial graph. Therefore, the treewidth of a clique of size \( n \), is always \( n - 1 \).

As for knowledge compilation, one typically uses the treewidth of some graph abstractions that characterize structural properties of a CNF. We will now review the most commonly used graph abstractions, namely, primal, dual, and incidence graphs.

The **primal graph** of a CNF is obtained by treating CNF variables as graph vertices, while adding an edge between two variables iff they appear in the same clause. The **dual graph** is obtained by treating CNF clauses as graph vertices, while adding an edge between two clauses iff they share a common variable. The **incidence graph** is obtained by treating CNF variables and clauses as graph vertices, while adding an edge between a variable and a clause iff the variable appears in the clause. For instance, Figure 5.2 depicts the primal, dual, and incidence graphs of the CNF \((X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)\).

In the sequel, we will use \( \text{twp} \), \( \text{twd} \), and \( \text{twi} \) to denote the treewidth of primal, dual, and incidence graphs of CNFs, respectively. It is known that \( \text{twp} \) and \( \text{twd} \) are incomparable, in the sense that there are classes of CNFs for which one can be bounded while the other is unbounded. For instance, consider the class of CNFs \( \Delta^1_n = \bigwedge_{i=2}^n X_1 \lor X_i \) defined over \( n \) variables \((n \geq 2)\). The primal treewidth of the CNFs in this class is 1, whereas the dual treewidth is \( n - 2 \). On the other hand, consider the class of CNFs \( \Delta^2_n = \bigvee_{i=1}^n X_i \) defined over \( n \) variables \((n \geq 1)\). The dual treewidth of the CNFs in this class is 0, whereas the primal treewidth is \( n - 1 \). Moreover, it has been shown that \( \text{twi} \leq \text{twp} + 1 \) and \( \text{twi} \leq \text{twd} + 1 \) for any CNF [KV00]. So, incidence treewidth dominates both primal and dual treewidth.
Figure 5.2: The primal, dual, and incidence graphs of the CNF \((X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)\).

In fact, this domination is strict as there exists CNF classes with bounded \(twi\) and unbounded \(twp\) and \(twd\). For instance, consider the class of CNFs \(\Delta^3_n = \Delta^1_n \land \Delta^2_n\). Indeed, the incidence treewidth of the CNFs in this class is 1, whereas both the primal and dual treewidth is \(n - 1\).
Table 5.1: The cutsets of CNF \((X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)\) with respect to variable ordering \(X, Y, Z, Q\).

<table>
<thead>
<tr>
<th>Index</th>
<th>Cutset</th>
<th>Separator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>({X \lor Y, \neg X \lor Q})</td>
<td>({X})</td>
</tr>
<tr>
<td>2</td>
<td>({Y \lor \neg Z, \neg X \lor Q})</td>
<td>({X, Y})</td>
</tr>
<tr>
<td>3</td>
<td>({\neg X \lor Q})</td>
<td>({X})</td>
</tr>
</tbody>
</table>

5.1.2 Cutwidth and Pathwidth

We now turn our attention to cutwidth and pathwidth, which apply directly to a CNF and have been used to bound the size of OBDDs obtained from CNFs [HD04].

We start with the definitions of cutwidth and pathwidth based on [HD04].

**Definition 13** (Cutwidth). Let \(\pi = V_1, \ldots, V_n\) be an ordering of the variables in CNF \(\Delta\). The \(i^{th}\) cutset of order \(\pi\) is the set of clauses in \(\Delta\) that mentions a variable \(V_j\), \(j \leq i\), and a variable \(V_k\), \(k > i\). The cutwidth of order \(\pi\) is the size of its largest cutset. The **cutwidth** of CNF \(\Delta\) is the smallest cutwidth attained by any variable ordering \(\pi\).

**Definition 14** (Pathwidth). Let \(\pi = V_1, \ldots, V_n\) be an ordering of the variables in CNF \(\Delta\). The \(i^{th}\) separator of order \(\pi\) is the set of variables \(V_j\), \(j \leq i\), that appear in the \(i^{th}\) cutset of order \(\pi\). The pathwidth of order \(\pi\) is the size of its largest separator. The **pathwidth** of CNF \(\Delta\) is the smallest pathwidth attained by any variable ordering \(\pi\).

For instance, consider the CNF \(\Delta = (X \lor Y) \land (Y \lor \neg Z) \land (\neg X \lor Q)\) and the variable ordering \(\pi = X, Y, Z, Q\). Table 5.1 shows the cutsets and separators of \(\pi\). Hence, both the cutwidth and pathwidth of order \(\pi\) is 2.

We note that cutwidth and pathwidth are incomparable. For example, the class of CNFs \(\Delta_1^n\) has bounded pathwidth but unbounded cutwidth. On the other hand, the class of CNFs \(\Delta_2^n\) has bounded cutwidth but unbounded pathwidth.
5.2 A Comparison of New and Classical CNF Parameters

In this section, we will compare new CNF parameters that we introduced in this thesis (i.e., CV-width, decision-width, and linear CV-width) with the classical CNF parameters reviewed in the previous section. The results of the comparison will lead us to new complexity bounds for knowledge compilation, which will be discussed in detail in the next section.

5.2.1 CV-Width and Incidence Treewidth

We will now show that CV-width strictly dominates the treewidth of the CNF incidence graph. For that, we first show that CV-width dominates the incidence treewidth (the proof is given in the appendix).

**Theorem 5.** Let $\Delta$ be a CNF whose incidence graph has treewidth $w$. There exists a vtree for CNF $\Delta$ whose CV-width is no greater than $w$.

We next show that the above dominance is strict. That is, there exists a class of CNFs whose incidence graph has unbounded treewidth, yet its CV-width is bounded. For that, we will consider $\Delta_n = \{C_1, \ldots, C_n\}$, where $C_i = X_1 \lor \ldots \lor X_i$.

In particular, we will show that its incidence treewidth is no less than $n/2 - 1$, yet its CV-width is 0.

We start with the incidence treewidth. Let $G$ be the incidence graph of $\Delta_n$. Let $I = \{X_1, \ldots, X_{\lfloor n/2 \rfloor+1}\}$ and let $S = \{C_{\lfloor n/2 \rfloor+1}, \ldots, C_n\}$. Note that each element of $I$ has an edge to each element of $S$ in graph $G$. Now, let $\pi = v_1, \ldots, v_{2n}$ be an elimination order for $G$. Let $v_i$ be the element of $I$ such that there is no $v_j$, for $j < i$, belonging to $I$. Consider now the graph $G_i$, which is obtained after eliminating first $i - 1$ elements of $\pi$ from $G$. We now show that $v_i$ has at least $\lfloor n/2 \rfloor$ neighbours in $G_i$, which is done by considering two cases.

- Assume none of $v_1, \ldots, v_{i-1}$ belongs to $S$. Then each edge between $v_i$ and
elements of $S$ still exists in $G_i$. That is, $v_i$ has at least $\lceil n/2 \rceil$ edges in $G_i$.

- Assume some of $v_1, \ldots, v_{i-1}$ belongs to $S$. Then $v_i$ has an edge to each element of $I$, except itself. To see this, note that each element of $I$ must exist in $G_i$ (because no $v_j, j < i$, belongs to $I$), and all elements of $I$ must become connected after eliminating an element from $S$ in one of first $i - 1$ eliminations. So, $v_i$ has at least $\lceil n/2 \rceil$ neighbors in $G_i$.

Thus, the width of order $\pi$ is no less than $\lceil n/2 \rceil$, which implies that the incidence treewidth is no less than $n/2 - 1$.

We next show the CV-width of $\Delta_n$ is 0. Consider the right-linear vtree induced by the variable ordering $X_1, \ldots, X_n$. Consider a vtree node $v$ whose left child is $X_i$. Since $v$ is a Shannon node, its cutset is empty. Let $\Gamma$ be the context clauses of $v$. If $i = 1$, then $\Gamma$ is empty and the width of $v$ is 0. Otherwise, $\Gamma = \{C_i, \ldots, C_n\}$.

Let $X$ be the variables inside $v$, and let $V$ be the context variables of $v$. Then, $\text{CNFs}(\Gamma, X|V) = \{\emptyset, \{X_i, X_i \lor X_{i+1}, \ldots, X_i \lor \ldots \lor X_n\}\}$. The width of $v$ is then 1. The CV-width of the vtree is then 0, which completes the proof.

**Theorem 6.** The incidence graph of a CNF may have an unbounded treewidth, yet its CV-width may be bounded.

Since incidence treewidth strictly dominates primal and dual treewidth, our result above immediately implies that CV-width also strictly dominates primal and dual treewidth.

### 5.2.2 Decision-Width and Primal Treewidth

We will now compare decision-width with the treewidth of the CNF primal graph, and show that decision-width strictly dominates the primal treewidth. First, we show that decision-width dominates the primal treewidth (the proof is given in the appendix).
Theorem 7. Let $\Delta$ be a CNF whose primal graph has treewidth $w$. There exists a decision vtree for CNF $\Delta$ whose decision-width is no greater than $w$.

We next show that decision-width can be bounded when primal treewidth is unbounded (i.e., the above dominance is strict). For that, we will consider $\Delta^2_n = \bigvee_{i=1}^n X_i$. In particular, we will show that its primal treewidth is $n - 1$, yet its decision-width is 0.

We start with the primal treewidth. Note that the primal graph of $\Delta_n$ is a complete graph, and complete graphs are known to have unbounded treewidth, which is $n - 1$ in this case.

We now consider decision-width. Consider the right-linear vtree induced by the variable ordering $X_1, \ldots, X_n$, which must be a decision vtree for $\Delta^2_n$. Consider a vtree node $v$ whose left child is $X_i$. Let $\Gamma$ be the context clauses of $v$. If $i = 1$, then $\Gamma$ is empty and the width of $v$ is 0. Otherwise, $\Gamma = \{X_1 \lor \ldots \lor X_n\}$. Let $Y$ be the variables outside $v$. Then, $CCNFs(\Gamma, Y) = \{\emptyset, \{X_i \lor \ldots \lor X_n\}\}$. The width of $v$ is then 1, which implies the decision-width of the vtree is 0, which completes the proof.

Theorem 8. The primal graph of a CNF may have an unbounded treewidth, yet its decision-width may be bounded.

5.2.3 Linear CV-Width, Cutwidth, and Pathwidth

We now turn our attention to cutwidth and pathwidth, which will be compared to linear CV-width. We will show that linear CV-width strictly dominates both cutwidth and pathwidth.

We first show that linear CV-width dominates cutwidth. That is, linear CV-width is no greater than cutwidth. Consider a CNF $\Delta$ and an ordering $\pi = V_1, \ldots, V_n$ of its variables. Let $w$ be the cutwidth of $\pi$. We will show that there exists a right-linear vtree whose linear CV-width is no greater than $w$. For that,
we will consider the right-linear vtree induced by $\pi$. Let $v$ be an internal vtree whose left child is $V_{i+1}$. It suffices to show that the width of $v$ is no greater than $w$. Let $\Gamma$ be the context clauses of $v$ and $Y$ be the variables outside $v$. Note that $|CCNFs(\Gamma, Y)| \leq 2^{|\Gamma|}$, hence $\log |CCNFs(\Gamma, Y)| \leq |\Gamma|$. Yet, as we will show next $\Gamma$ equals the $i^{th}$ cutset of order $\pi$, which implies the width of $v$ is no greater than $w$. To see this, note that $Y = \{V_1, \ldots, V_i\}$ and the variables inside $v$ are $\{V_{i+1}, \ldots, V_n\}$, making the context clauses the same as ones in the $i^{th}$ cutset of $\pi$.

We now show that linear CV-width dominates pathwidth. That is, linear CV-width is no greater than pathwidth. For that, we will consider the same setting used for cutwidth above, except that $w$ will represent the pathwidth of order $\pi$. Since $\Gamma$ equals the $i^{th}$ separator of $\pi$, $Vars(\Gamma) \cap Y$ equals the $i^{th}$ separator of order $\pi$. Further, since $|CCNFs(\Gamma, Y)| \leq 2^{|Vars(\Gamma) \cap Y|}$, we have $\log |CCNFs(\Gamma, Y)| \leq |Vars(\Gamma) \cap V|$, and thus the width of $v$ is no greater than $w$.

**Theorem 9.** Let $\Delta$ be a CNF whose cutwidth is $cw$ and pathwidth is $pw$. There exists a right-linear vtree for CNF $\Delta$ whose linear CV-width is no greater than $cw$ and $pw$.

We now know that linear CV-width dominates both cutwidth and pathwidth. We next demonstrate that the latter widths can be unbounded when the former is bounded. For that, we will consider $\Delta_n = \{X \lor Y_1, \ldots, X \lor Y_n, Y_1 \lor \ldots \lor Y_n\}$, which is defined over $n + 1$ variables and $n + 1$ clauses ($n \geq 1$). We will show that its cutwidth is no less than $n/2 - 1$, that its pathwidth is no less than $n - 2$, and that its linear CV-width is no greater than 1.

We will first show that the cutwidth of $\Delta_n$ is unbounded. Consider a variable ordering $\pi = V_1, \ldots, V_{n+1}$ for $\Delta_n$. To show that its cutwidth is no less than $n/2 - 1$, we will look at the position of variable $X$ in the order $\pi$. Assume that $X$ appears at index $i$ (that is, $V_i = X$), and consider the following two cases.

- If index $i$ happens to be no greater than $\lfloor n/2 \rfloor$, then the set $\{V_{i+1}, \ldots, V_{n+1}\}$
has at least \(\lceil n/2 \rceil + 1\) variables. Note that there is a distinct clause \(\{X \lor V_j\}\) for each \(V_j, j \neq i\). So, the \(i\)th cutset has at least \(\lceil n/2 \rceil + 1\) clauses.

- If index \(i\) happens to be no less than \(\lfloor n/2 \rfloor + 1\), then the set \(\{V_1, \ldots, V_{i-1}\}\) has at least \(\lfloor n/2 \rfloor\) variables. Then, the \((i - 1)\)th cutset has at least \(\lfloor n/2 \rfloor\) clauses (for the same reason as above).

Therefore, the cutwidth is no less than \(\lfloor n/2 \rfloor\), which is no less than \(n/2 - 1\), for any order \(\pi\).

To show that the pathwidth is unbounded for any order \(\pi\), we note that the size of the \((n - 1)\)th separator of any order \(\pi\) must be no less than \(n - 2\). To see this, note that one of the last two variables in any order \(\pi\) will be different than \(X\). So, the clause \(\{Y_1 \lor \ldots \lor Y_n\}\) must appear in the \((n - 1)\)th cutset of order \(\pi\), which implies that the \((n - 1)\)th separator must contain at least \(n - 2\) variables. Thus, we can conclude that the pathwidth is no less than \(n - 2\) for any order \(\pi\).

Finally, we will show that the linear CV-width of \(\Delta_n\) is no greater than 1. Consider the variable ordering \(\pi = X, Y_1, \ldots, Y_n\) and the right-linear vtree induced by \(\pi\). Figure 5.3 depicts this vtree with the corresponding CCNFs(\(\Gamma, Y\)) of each internal node. Accordingly, the linear CV-width of this vtree is 1, and hence the linear CV-width of CNF \(\Delta_n\) is no greater than 1.

**Theorem 10.** The cutwidth of a CNF may be unbounded treewidth, yet its linear CV-width may be bounded. Similarly, the pathwidth of a CNF may be unbounded treewidth, yet its linear CV-width may be bounded.

### 5.3 A Discussion on the Comparison

In this section, we will discuss the results of the comparison made in the previous section. In particular, we will provide new upper bounds on the compilation sizes of DNNFs and its subsets.
Figure 5.3: A right-linear vtree induced by order $X, Y_1, \ldots, Y_n$. Nodes $v$ show $CCNF_s(\Gamma, Y)$ for $\Delta_n = \{X \lor Y_1, \ldots, X \lor Y_n, Y_1 \lor \ldots \lor Y_n\}$, where $\Gamma$ is context clauses and $Y$ is variables outside $v$.

Table 5.2 lists the NNF languages, the corresponding classical CNF parameters and the new CNF parameters that bound the compilation sizes.

Accordingly, two algorithms for compiling structured DNNFs were given in [PD10a]. One algorithm splits on variables and the other one splits on clauses. The latter has a time and space complexity that is exponential in the treewidth of the CNF dual graph, and the former has a time and space complexity that is exponential in the treewidth of the CNF primal graph. Algorithm 1 (proposed in Chapter 3) splits on both variables and clauses. One would have expected that this combination will lead to a complexity that is a minimum of the two complexities attained by the mentioned algorithms. Interestingly though, the combination leads to a more significant improvement. In particular, our algorithm has a time and space complexity that is exponential in CV-width, which we showed to strictly dominate the treewidth of the CNF incidence graph. Moreover, it is already known that this treewidth dominates the ones for the CNF primal and dual graphs.
Another bound was recently shown for DNNFs compiled from CNFs [RP13]. Given a CNF with \( n \) variables, size \( m \), and incidence treewidth \( w \), this bound shows that the DNNF size is in \( O((n + m)^{3w}) \). Algorithm 1 improves on this bound in two fundamental ways. First, our bound applies to structured DNNF, which is a subset of DNNF that supports a polytime conjoin operation (not supported by unstructured DNNF). Second, our bound is based on CV-width, which strictly dominates the treewidth of the incidence graph. Hence, our bound significantly improves on the existing bound for DNNFs, even when unstructured. Finally, our size upper bound is linear in the number of variables, whereas the existing upper bound is linear in the number of variables plus the size of the CNF (which can be much larger than the number of variables).

The existing size upper bounds on d-DNNFs, Decision-DNNFs, and SDDs are all exponential in the treewidth of the CNF primal graph [Dar01a, HD07, Dar11]. For these three languages, we showed a tighter bound that is exponential only in Decision-width, which strictly dominates the primal treewidth (Algorithm 2 and Algorithm 3 proposed in Chapter 4). Indeed, this result not only improves the existing complexity results for knowledge compilation but also an existing result on the complexity of model counting (was also exponential in the CNF primal treewidth).

<table>
<thead>
<tr>
<th>Language</th>
<th>Classical Parameter</th>
<th>New Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNNF</td>
<td>incidence treewidth</td>
<td>CV-width</td>
</tr>
<tr>
<td>Structured DNNF</td>
<td>primal and dual treewidth</td>
<td>CV-width</td>
</tr>
<tr>
<td>d-DNNF</td>
<td>primal treewidth</td>
<td>Decision-width</td>
</tr>
<tr>
<td>Decision-DNNF</td>
<td>primal treewidth</td>
<td>Decision-width</td>
</tr>
<tr>
<td>SDD</td>
<td>primal treewidth</td>
<td>Decision-width</td>
</tr>
<tr>
<td>OBDD</td>
<td>cutwidth and pathwidth</td>
<td>Linear CV-width</td>
</tr>
</tbody>
</table>

Table 5.2: Complexity parameters bounding the sizes of DNNF and its subsets.
An algorithm for compiling OBDDs was presented in [HD04]. The complexity of the algorithm is exponential in the cutwidth or the pathwidth of input CNF. Algorithm 3 is exponential in the linear CV-width of the CNF. Since linear CV-width strictly dominates both cutwidth and pathwidth, our upper bound significantly improves on the ones given in [HD04].
5.A Proof of Theorem 5

In this section, we will prove Theorem 5. For that, we will use another auxiliary structure and its associated notion of width, which is based on [Dar01a]. We start defining necessary concepts.

**Definition 15.** Let $G$ be the incidence graph of a CNF. An incidence dtree for the CNF is a full binary tree, whose leaves have a one-to-one correspondence with the edges of $G$.

Note that an incidence dtree node contains both variables and clauses under the subtree rooted at itself. Being able to refer those variables and clauses separately will be useful:

$$Vars(d) = \begin{cases} 
\text{Variable of the node,} & \text{if } d \text{ is a leaf node}, \\
Vars(d^l) \cup Vars(d^r), & \text{otherwise}.
\end{cases}$$

$$CNF(d) = \begin{cases} 
\text{Clause of the node,} & \text{if } d \text{ is a leaf node}, \\
CNF(d^l) \cup CNF(d^r), & \text{otherwise}.
\end{cases}$$

We will also need to represent variables and clauses of an incidence dtree node together. For such cases, we will use the following notation:

$$Labels(d) = Vars(d) \cup CNF(d).$$

We will now provide some definitions to define the width of an incidence dtree.

**Definition 16.** The cutset of an internal incidence dtree node $d$ is

$$Cutset(d) = (Labels(d^l) \cap Labels(d^r)) \setminus Acutset(d),$$

where $Acutset(d)$ is the union of cutsets of ancestors of $d$. 
Definition 17. The context of an incidence dtree node $d$ is

$$\text{Context}(d) = \text{Labels}(d) \cap \text{Acutset}(d).$$

where Acutset($d$) is the union of cutsets of ancestors of $d$.

Definition 18. The cluster of an incidence dtree node $d$ is

$$\text{Cluster}(d) = \begin{cases} 
\text{Labels}(d), & \text{if } d \text{ is a leaf node,} \\
\text{Cutset}(d) \cup \text{Context}(d), & \text{otherwise.}
\end{cases}$$

We are now ready to define the width of an incidence dtree:

Definition 19. The width of an incidence dtree is the size of its maximal cluster minus one.

We next relate the incidence treewidth with the incidence dtree width.

Theorem 11. Given a CNF whose incidence graph has treewidth $w$, we can construct an incidence dtree of width $w$.

Proof. Consider a CNF $\Delta$ whose incidence graph has treewidth $w$. Then, we can create an auxiliary CNF $\Gamma$ from $\Delta$ as follows:

- for each variable $V$ in $\Delta$, add a variable in $\Gamma$;
- for each clause $C$ in $\Delta$, add a variable in $\Gamma$;
- add a binary clause in $\Gamma$ for each var-variable $V$ and clause-variable $C$ when a literal of variable $V$ appears in clause $C$.

Note that the primal graph of $\Gamma$ is identical to the incidence graph of $\Delta$. So, the primal graph of $\Gamma$ has treewidth $w$. We also know that we can create a "dtree" for $\Gamma$, a full binary tree whose leaves are the clauses of $\Gamma$, which has width same as the treewidth of the primal graph of $\Gamma$ [Dar01a]. Finally, such a "dtree" of $\Gamma$ is actually an incidence dtree for $\Delta$, and their widths are the same (since both definitions of widths are the same), which is $w$. ■
Now we know that we can create an incidence dtree for a CNF whose width is the same as the treewidth of CNF incidence graph, we will show a width-preserving algorithm that can construct a vtree from an incidence dtree. From here on, we will call an incidence dtree simply as a dtree. We next introduce some more definitions.

**Definition 20.** Consider a CNF, a corresponding vtree, and an internal vtree node \(v\). Let \(\alpha\) be a context clause of \(v\). Then, \(\alpha\) is called **Type I context clause** iff all variables of \(\alpha\) that are outside \(v\) are context variables of \(v\). Otherwise, \(\alpha\) is a **Type II context clause**.

**Definition 21.** Let \(\Delta\) be a CNF and \(V\) be a set of variables. The **cardinality** of \((\Delta, V)\) is defined as

\[
\text{Card}(\Delta, V) = \arg \min_{\Gamma \subseteq \Delta} |\Gamma| + |\text{Vars}(\Delta \setminus \Gamma) \cap V|.
\]

The notion of cardinality provides an upper bound on the number of distinct CNFs that can be obtained from a given set of clauses after conditioning those clauses on given variables (i.e., bounding the size of \(CCNF(\Delta, V)\)). For instance, consider the following pair of CNF and variables:

\[
\Delta = \{X \lor Y, \neg X \lor \neg Y, X \lor \neg Q, \neg X \lor Q, X \lor Z \lor W\} \text{ and } V = \{Q, W, Y, Z\}.
\]

Suppose we want to bound the number of distinct CNFs we can obtain by conditioning \(\Delta\) on all (complete) variable instantiations of \(V\). Note here that, to compute \(\text{Card}(\Delta, V)\), we should go over the subsets \(\Gamma\) of \(\Delta\) and variables \(\text{Vars}(\Delta \setminus \Gamma) \cap V\). The table below shows an incomplete list of such pairs of clauses and variables.

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>{X \lor Y, \neg X \lor \neg Y, X \lor \neg Q}</td>
<td>(Q, W, Y, Z)</td>
</tr>
</tbody>
</table>

Back to bounding the distinct number of CNFs, given a variable instantiation over \(\{Q, W, Y, Z\}\), observe that any clause in \(\Delta\) would be either subsumed or shrunken to a clause over variable \(X\). Then, one (loose) upper bound is \(2^5\), as there are 5 clauses. In fact, this is exactly what the 1\(^{st}\) row in the table represents.
Another way to get an (still loose) upper bound is to count the number of variable instantiations, as each instantiation may create a different CNF. So, the bound is $2^4$, as there are 4 variables to be conditioned on. In this case, this is what the 2nd row in the table represents. So, we have a better upper bound. However, we may even get better bounds by considering a subset of clauses and a subset of variables, that is what other rows in the table represent. Consider the 3rd row in the table. It essentially tells that instantiations over variables \{Q,Y\} can bound the number of distinct CNFs obtained from all clauses but $X \lor Z \lor W$, by conditioning on \{Q,W,Y,Z\}. So, there are at most $2^2$ such distinct CNFs.

As we can get two distinct CNFs from the clause $X \lor Z \lor W$ by conditioning on \{Q,W,Y,Z\}, which are $\top$ and \{X\}, the upper bound is $2^2 \cdot 2$. Note that the 3rd row contains the minimum $\Gamma$ in the definition of $\text{Card}(\Delta, V)$. In the light of this example, we have the following result:

**Theorem 12.** Given a CNF $\Delta$ and a set of variables $V$, let $\#\text{CNF}(\Delta, V) = |\text{CCNFs}(\Delta, V)|$. If $k$ is the cardinality of $(\Delta, V)$, then $\#\text{CNF}(\Delta, V) \leq 2^k$.

We will now show the relationship between $\#\text{CNFs}(\Gamma, X|V)$ and the cardinality, for which we need the following three lemmas.

**Lemma 6.** Let $\Delta(X, V)$ be a CNF over two disjoint sets of variables $X$ and $V$. Then, $|\text{CNFs}(\Delta, X|V)|$ is equal to the number of distinct CNFs $\Delta|v$.
Proof. Since clauses in $\Delta$ are over variables $X$ and $V$, $\Delta|v$ is a CNF over variables $X$. This implies that the set of clauses in $\Delta|v$ that mention variables inside and outside $X$, is empty. Thus, we have the following equations, which show that $|CNFs(\Delta, X|V)|$ is equal to the number of distinct CNFs $\Delta|v$.

$$CNFs(\Delta, X|V) = \bigcup_{v} CNFs(\Delta|v, X) = \bigcup_{v} \{\Delta|v \cup \Gamma \mid \Gamma \subseteq \{\}\} = \bigcup_{v} \{\Delta|v\}.$$

Lemma 7. Let $\Delta$ be a CNF, and $X$ and $V$ be two disjoint sets of variables. Then, $|CNFs(\Delta, X|V)| \leq 2^{|\Delta|}$.

Proof. Let $\Gamma$ be a CNF belonging to $CNFs(\Delta, X|V)$, and $\alpha$ be a clause of $\Gamma$. Then, $\alpha$ must be over variables $X$. In fact, $\alpha$ is the sub-clause of some clause $\beta$ in $\Delta$, which is obtained by replacing $\beta$ by its sub-clause that mentions variables $X$. To see this, note that $\alpha$ can be obtained after performing a (possible) variable splitting and a (possible) clause splitting on $\Delta$. This implies that $\Gamma$ is a CNF that can be constructed using clauses of $\Delta$. As the number of different CNFs one can construct using clauses of $\Delta$ is no greater than $2^{|\Delta|}$, the lemma holds.

Lemma 8. Let $\Delta = \Delta_1 \cup \Delta_2$ be a CNF, and $X$ and $V$ be two disjoint sets of variables. Then, $|CNFs(\Delta, X|V)| \leq |CNFs(\Delta_1, X|V)| \times |CNFs(\Delta_2, X|V)|$.

Proof. In the following, for any CNF $\Sigma$, we denote the clauses in $\Sigma$ that only mention variables $X$ by $\Sigma(X)$. Further, we assume $\{\delta_1, \ldots, \delta_n\}_v$ contains the clauses in $\Delta|v$ that mention variables inside and outside $X$, where $v$ is an instantiation of $V$. So, we denote by $\{\alpha_1, \ldots, \alpha_n\}_v$ the sub-clauses of $\delta_i$ with variables in $X$. Finally, $\{\gamma_1^1, \ldots, \gamma_k^1\}_v$ denotes the maximal subset of $\{\alpha_1, \ldots, \alpha_n\}_v$ such that each $\gamma_i^1$ appears in $\Delta_1|v$, and likewise $\{\gamma_1^2, \ldots, \gamma_l^2\}_v$ is the maximal subset of $\{\alpha_1, \ldots, \alpha_n\}_v$ such that each $\gamma_i^2$ appears in $\Delta_2|v$. Then, we have the following:
\[ CNFs(\Delta, X|V) = \bigcup_v CNFs(\Delta|v, X) \]
\[ = \bigcup_v \{ \Delta|v(X) \cup \Gamma \mid \Gamma \subseteq \{\alpha_1, \ldots, \alpha_n\}_v \} \]
\[ = \bigcup_v \{ \Delta_1|v(X) \cup \Delta_2|v(X) \cup \Gamma \mid \Gamma \subseteq \{\alpha_1, \ldots, \alpha_n\}_v \} \]
\[ = \bigcup_v \{ \Delta_1|v(X) \cup \Gamma_1 \cup \Delta_2|v(X) \cup \Gamma_2 \mid \Gamma_1 \subseteq \{\gamma^1_1, \ldots, \gamma^1_l\}_v, \Gamma_2 \subseteq \{\gamma^2_1, \ldots, \gamma^2_l\}_v \} \]
\[ = \bigcup_v \{ \Sigma_1 \cup \Sigma_2 \mid \Sigma_1 \in CNFs(\Delta_1|v, X), \Sigma_2 \in CNFs(\Delta_2|v, X) \} \]

Note that \( \Sigma_1 \) belongs to \( CNFs(\Delta_1|v, X) \) implies that \( \Sigma_1 \) also belongs to \( CNFs(\Delta_1, X|V) \) (analogously, \( \Sigma_2 \) belongs to \( CNFs(\Delta_2, X|V) \)). Therefore, \(|CNFs(\Delta, X|V)| \leq |CNFs(\Delta_1, X|V)| \times |CNFs(\Delta_2, X|V)|\).}

\textbf{Theorem 13.} Let \( v \) be an internal vtree node, with variables \( X \), context clauses \( \Gamma \), Type I context clauses \( \Gamma_1 \), Type II context clauses \( \Gamma_2 \), and context variables \( V \). Then, \(#CNFs(\Gamma, X|V) \leq |\Gamma_2| + Card(\Gamma_1, V)\).

\textbf{Proof.} Clauses in \( \Gamma_1 \) are over variables \( X \) and \( V \). So, by Lemma 6 and Theorem 12, \(|CNFs(\Gamma_1, X|V)| \leq 2^{Card(\Gamma_1, V)}\). Also, by Lemma 7, \(|CNFs(\Gamma_2, X|V)| \leq 2^{\Gamma_2}|. Finally, by Lemma 8, \(|CNFs(\Gamma, X|V)| \leq |CNFs(\Gamma_1, X|V)| \times |CNFs(\Gamma_2, X|V)|\). Therefore, \(|CNFs(\Gamma, X|V)| \leq 2^{Card(\Gamma_1, V)\Gamma_2}|. By taking logs of the both sides, we conclude \(#CNFs(\Gamma, X|V) \leq |\Gamma_2| + Card(\Gamma_1, V).\)

We next present a width-preserving algorithm that constructs vtrees from dtrees. In particular, given a dtree of width \( w \), we will create a vtree of width at most \( w \).
Algorithm 4: d2v(d)

Input: d : an incidence dtree node.

Output: A vtree or nil.

if d is a leaf then
  if Vars(d) appears only in CNF(d) then
    return Leaf vnode labeled with Vars(d)
  else return nil

C ← cutset(d)

T ← right-linear vtree obtained from Vars(C)

T_l ← d2v(d_l)
T_r ← d2v(d_r)

if T_l and T_r are nil then return T

else if T_l and T_r are not nil then
  T' ← vtree node whose left child is T_l and right child is T_r
  return T by making T' its right most child

else if T_l is nil and T_r is not nil then
  return T by making T_r its right most child

else if T_l is not nil and T_r is nil then
  return T by making T_l its right most child

Given a dtree for a CNF, Algorithm 4 computes a vtree. Observe that in the algorithm, an internal vtree node can be constructed on either Line 6 or Line 11. Also, any such node constructed on Line 6 is a Shannon node.

In the following, we will show the result relating the width of a vtree to the width of a dtree that constructs the vtree, for which we need the following five lemmas.

The first lemma shows the relationship between the leaf nodes of a dtree node d and an internal vtree node v which is constructed during a call d2v(d).
Lemma 9. Let \( v \) be an internal vtree node constructed by Algorithm 4 on a call \( d2v(d) \). Let \( X \) be a variable inside \( v \) and \( C \) be a clause which contains a literal of \( X \). Then there is a leaf dtree node with label \( \{X, C\} \) in the subtree rooted at \( d \).

Proof. Since \( v \) is constructed at the call \( d2v(d) \) and \( X \) is inside \( v \), the leaf vtree node \( v' \) with label \( X \) must be constructed at a call \( d2v(d') \) where \( d' \) is a dtree node in the subtree rooted at \( d \). We show that there is a leaf dtree node with label \( \{X, C\} \) in the subtree rooted at \( d' \). As \( v' \) is a leaf, it is constructed on either Line 2 or Line 6. Assume \( v' \) is constructed on Line 2. Then, by the if statement, \( C \) is the only clause in which a literal of \( X \) appears and also \( d' \) is the leaf dtree node labeled with \( \{X, C\} \). Now, assume \( v' \) is constructed on Line 6. Then, \( X \) is in the cutset of \( d' \), which implies \( X \) is in \( Labels(d') \). Moreover, by Definition 15 (Dtree), there is a leaf node \( l \) labeled by \( \{X, C\} \) in the dtree in which \( d' \) appears. Assume \( l \) is outside \( d' \). Since \( X \) is in \( Labels(d') \), \( X \) must be in the cutset of an ancestor of \( d' \). But, this implies that \( X \) cannot be in the cutset of \( d' \), which is a contradiction. Thus, \( l \) is inside \( d' \). So, there is a leaf dtree node \( l \) with label \( \{X, C\} \) inside \( d' \). Because \( d' \) is inside \( d \), \( l \) is also inside \( d \). \( \blacksquare \)

The next two lemmas show the relationship between clauses outside of a cluster of a dtree node \( d \) and an internal vtree node \( v \) which is constructed during a call \( d2v(d) \).

Lemma 10. Let \( v \) be an internal vtree node constructed by Algorithm 4 at a call \( d2v(d) \). Let \( C \) be a clause mentioning variables inside and outside \( v \). If \( C \) is not in \( Cluster(d) \), then \( Vars(C) \cap Vars(v) \) is in \( Cluster(d) \).

Proof. Assume \( C \) is not in \( Cluster(d) \). Let \( X \) be a variable inside \( v \) such that a literal of \( X \) appears in \( C \). Let \( Y \) be a variable outside \( v \) such that a literal of \( Y \) appears in \( C \). By Lemma 9, there is a leaf dtree node \( l_x \) with label \( \{X, C\} \) in the subtree rooted at \( d \). Also, by Definition 15 (Dtree), there is a leaf dtree node \( l_y \) with label \( \{Y, C\} \) in the dtree in which \( d \) appears. Since \( l_x \) is inside \( d \), \( l_y \) must
also be inside \( d \). Otherwise, \( C \) is in \( \text{Cluster}(d) \), which is a contradiction. Wlog, assume that \( l_y \) is inside \( d' \). Now assume that \( Y \) is not in \( \text{Cluster}(d) \). It implies that \( Y \) only appears in \( d' \). Recall that \( v \) is constructed at the call \( d2v(d) \), and \( Y \) is outside \( v \). However, if \( Y \) only appears in \( d' \), then it can be constructed only at a call \( d2v(d') \), where \( d' \) is a dtree node in the subtree rooted at \( d' \). This means \( Y \) is inside \( v \), which is a contradiction. So, \( Y \) is in \( \text{Cluster}(d) \). Thus, when \( C \) is not in \( \text{Cluster}(d) \), we have \( \text{Vars}(C) \cap \overline{\text{Vars}(v)} \) is in \( \text{Cluster}(d) \). ■

Lemma 11. Let \( v \) be an internal vtree node constructed by Algorithm 4 at a call \( d2v(d) \). Let \( C \) be a clause mentioning variables inside and outside \( v \). If \( C \) is not in \( \text{Cluster}(d) \), then \( C \) is a Type I context clause of \( v \).

Proof. Assume \( C \) is not in \( \text{Cluster}(d) \). We need to show that all variables of \( C \) appearing outside \( v \) are context variables of \( v \), and \( C \) is not in the cutset of \( v \).

Let \( Y \) be a variable of \( C \) appearing outside \( v \). By Lemma 10, \( \text{Vars}(C) \cap \overline{\text{Vars}(v)} \) is in \( \text{Cluster}(d) \). So, \( Y \) is in \( \text{Cluster}(d) \). It implies \( Y \) is in the cutset of a dtree node \( d' \) that is on the path from the root of dtree to \( d \) (including \( d \)). So, the leaf vtree node labeled as \( Y \) must be created at the call \( d2v(d') \) on Line 6. As \( d' \) is either \( d \) or an ancestor of \( d \), and \( Y \) is outside \( v \), \( Y \) is a context variable of \( v \). Let \( \Delta \) be the cutset of \( v \). We now show \( C \) is not in \( \Delta \). Assume \( C \) is in \( \Delta \). Then, \( v \) cannot be a Shannon node, since \( \Delta \) is empty for Shannon nodes, by Definition 6 (Cutset). So, \( v \) must be created on Line 11. Also, by Definition 6 (Cutset), there is a variable \( X_l \) inside \( v^l \) such that a literal of \( X_l \) appears in \( C \), and, similarly, there is a variable \( X_r \) inside \( v^r \) such that a literal of \( X_r \) appears in \( C \). Since \( v \) is constructed on Line 11, \( v^l \) is constructed in a call \( d2v(d') \) where \( d' \) is a dtree node in the subtree rooted at \( d' \). Then, by Lemma 9, there is a leaf dtree node \( l_1 \) with label \( \{X, C\} \) inside \( d' \). As \( d' \) is inside \( d' \), \( l_1 \) is also inside \( d' \). Analogously, there is a leaf dtree node \( l_2 \) with label \( \{Y, C\} \) inside \( d' \). Then, \( C \) is in \( \text{Cluster}(d) \), which is a contradiction. So, \( C \) is not in \( \Delta \). Hence, we conclude that \( C \) is a Type I context clause of \( v \). ■
The next two lemmas show the relationship between the cluster of a dtree node $d$ and the cutset and context clauses of an internal vtree node $v$ which is constructed during a call $d2v(d)$.

**Lemma 12.** Let $v$ be an internal vtree node with cutset $\Delta$ and Type II context clauses $\Gamma_2$, constructed by Algorithm 4 at a call $d2v(d)$. Then, $(\Delta \cup \Gamma_2) \subseteq \text{Cluster}(d)$.

**Proof.** Let $C$ be a clause in $(\Delta \cup \Gamma_2)$. If $C$ is not in $\text{Cluster}(d)$, then $C$ is a Type I context clause of $v$ by Lemma 11. However, by Definition 20 (Type I context clause), $C$ cannot be a Type I context clause of $v$. So, $C$ is in $\text{Cluster}(d)$. That is, $(\Delta \cup \Gamma_2) \subseteq \text{Cluster}(d)$. ■

**Lemma 13.** Let $v$ be an internal vtree node with cutset $\Delta$, Type I context clauses $\Gamma_1$, Type II context clauses $\Gamma_2$, and context variables $V$, constructed by Algorithm 4 at a call $d2v(d)$. Then, $\text{Card}(\Gamma_1, V) \leq |\text{Cluster}(d) \setminus (\Delta \cup \Gamma_2)|$.

**Proof.** Let $S = \text{Cluster}(d) \setminus (\Delta \cup \Gamma_2)$. Let $\Gamma$ be the maximal subset of $\Gamma_1$ such that $\Gamma \subseteq S$. We show that $\Gamma \cup (\text{Vars}(\Gamma_1 \setminus \Gamma) \cap V) \subseteq S$, which implies $\text{Card}(\Gamma_1, V) \leq |S|$. By definition, $\Gamma \subseteq S$. Let $C$ be a clause in $\Gamma_1 \setminus \Gamma$. So, $C$ is not in $S$. Also, as $\Gamma_1$ and $\Delta \cup \Gamma_2$ are disjoint, $C$ is not in $\text{Cluster}(d)$. Then, by Lemma 10, $\text{Vars}(C) \cap \overline{\text{Vars}(v)}$ is in $\text{Cluster}(d)$. In fact, as $\text{Vars}(C) \cap \overline{\text{Vars}(v)}$ and $\Delta \cup \Gamma_2$ are disjoint, $\text{Vars}(C) \cap \overline{\text{Vars}(v)}$ is in $S$. Also, by Definition 20 (Type I context clause), all variables of $C$ appearing outside $v$ are context variables of $v$. Thus, $\text{Vars}(C) \cap V$ is in $S$, which implies $(\text{Vars}(\Gamma_1 \setminus \Gamma) \cap V) \subseteq S$. As we have $\Gamma \cup (\text{Vars}(\Gamma_1 \setminus \Gamma) \cap V) \subseteq S$ and $\Gamma$ and $\text{Vars}(\Gamma_1 \setminus \Gamma) \cap V$ are disjoint, $|\Gamma| + |\text{Vars}(\Gamma_1 \setminus \Gamma) \cap V| \leq |S|$. So, by Definition 21, $\text{Card}(\Gamma_1, V) \leq |S|$.

We are now ready to show our main result, which relates the width of a vtree to the width of a dtree that constructs the vtree.

**Theorem 14.** Let $T$ be a vtree generated by Algorithm 4 with dtree $T'$ being input. Let $w$ be the CV-width of $T$ and $w'$ be the width of $T'$. Then, $w \leq w'$.
Proof. Let \( v \) be an internal node of \( T \) with cutset \( \Delta \), context clauses \( \Gamma \), Type I context clauses \( \Gamma_1 \), Type II context clauses \( \Gamma_2 \), and context variables \( V \), which is constructed at a call \( d2v(d) \), where \( d \) is a dtree node in \( T' \). By Lemma 13, \( \text{Card}(\Gamma_1, V) \leq |\text{Cluster}(d) \setminus (\Delta \cup \Gamma_2)| \). By Lemma 12, \( (\Delta \cup \Gamma_2) \subseteq \text{Cluster}(d) \). By Definition 20, \( \Delta \) and \( \Gamma_2 \) are disjoint. So, \( |\Delta| + |\Gamma_2| + \text{Card}(\Gamma_1, V) \leq |\text{Cluster}(d)| \). Then, By Theorem 13, \( |\Delta| + \#\text{CNFs}(\Gamma, X|V) \leq |\text{Cluster}(d)| \). Thus, we conclude, by noting Definition 19 and Definition 7, that \( w \leq w' \). ■

Theorem 5. Let \( \Delta \) be a CNF whose incidence graph has treewidth \( w \). There exists a vtree for CNF \( \Delta \) whose CV-width is no greater than \( w \).

Proof. By Theorem 11, we can construct a dtree for \( \Delta \) with width \( w \). Then, by Theorem 14, given that dtree as input, Algorithm 4 constructs a vtree for \( \Delta \) with CV-width \( \leq w \). ■

5.B Proof of Theorem 7

The proof is analogous to the proof of Theorem 5. Here, instead of using an incidence dtree to construct a vtree, we will use a dtree [Dar01a]. This is because we want to create a decision vtree. It is not hard to see that vtrees obtained from dtrees as in Algorithm 4 will indeed be a decision vtree (due to the property of cutsets). Moreover, it is known that the dtree width of a CNF is the same as the primal treewidth of the CNF [Dar01a]. So, it is enough to show that the width of the constructed vtree is no more than the width of the dtree used to construct the vtree. This boils down to show that \( \log |\text{CNFs}(\Gamma, V)| \leq |\text{Cluster}(d)| \), where \( d \) is a dtree node used to construct a vtree node whose context clauses are \( \Gamma \) and variables are \( V \). We already know, by Theorem 12, that \( \log |\text{CNFs}(\Gamma, V)| \leq \text{Card}(\Gamma, V) \). Therefore, it is enough to show that \( \text{Card}(\Gamma, V) \leq |\text{Cluster}(d)| \).

This can be done analogously to Lemma 13, where \( \Delta \) and \( \Gamma_2 \) will be empty and \( \Gamma \) will be equal to \( \Gamma_1 \) (due to having a decision vtree). Thus, we know that the
vtree width will be no more than the dtree width, which implies that the decision width will be no more than the primal treewidth.
CHAPTER 6

A Top-Down Compiler for Sentential Decision Diagrams

We will turn our attention in this chapter to developing practical knowledge compilers. In particular, we will identify a subset of SDDs, called Decision-SDDs, and introduce a top-down algorithm for compiling CNFs into Decision-SDDs. This algorithm is based on the one introduced in Chapter 4, but made practical by adding techniques from the SAT literature and defining a specific caching scheme. We will provide a formal description of the new compiler and empirically evaluate it, showing significant improvements in compilation time against the state-of-the-art bottom-up SDD compiler when the input is a CNF. We will also modify our algorithm to yield a new model counter, and perform an extensive empirical evaluation, comparing it to the state-of-the-art model counters. The material in this chapter is based on the work published in [OD15] and accepted in [OD17a].

6.1 Introduction

Since [DM02], the area of knowledge compilation has settled on three major research directions: (1) identifying new tractable representations that are characterized by their succinctness and polynomial time support for certain queries and transformations; (2) developing efficient knowledge compilers; and (3) using those representations and compilers in various applications, such as diagnosis [Bar05, EW06], planning [Bar04, PBD05, Hua06], and probabilistic infer-
This chapter focuses on developing efficient compilers. In particular, our emphasis is on the compilation of the sentential decision diagram (SDD) [Dar11] that was recently discovered as a tractable representation of Boolean functions. SDDs are a strict superset of ordered binary decision diagrams (OBDDs) [Bry86], which are one of the most popular, tractable representations of Boolean functions. Despite their generality, SDDs still maintain some key properties behind the success of OBDDs in practice. This includes canonicity, which leads to unique representations of Boolean functions. It also includes the support of an efficient `Apply` operation that combines SDDs using Boolean operators. SDDs also come with tighter size upper bounds than OBDDs [Dar11, OD14b, Raz14] and an exponential separation [Bov16]. Moreover, SDDs have been used in different applications, such as probabilistic planning [HB13], probabilistic inference [CKD13], verification of multi-agent systems [LP15], and tractable learning [KDC14, CDD15].

Almost all of these applications are based on the bottom-up SDD compiler developed by [CD13a], which was also used to compile CNFs into SDDs [CD13b]. This compiler constructs SDDs by compiling small pieces of a knowledge base (e.g., clauses of a CNF). It then combines these compilations using the `Apply` operation to build a compilation for the full knowledge base.

An alternative to bottom-up compilation is top-down compilation. This approach starts the compilation process with a full knowledge base. It then recursively compiles the fragments of the knowledge base that are obtained through conditioning. The resulting compilations are then combined to obtain the compilation of the full knowledge base. All existing top-down compilers assume CNFs as input, while bottom-up compilers can work on any input due to the `Apply` operation. Yet, compared to bottom-up compilation, top-down compilation has been previously shown to yield significant improvements in compilation time and space when compiling CNFs into OBDDs [HD04]. Thus, it has a potential to
further improve the results on CNF to SDD compilations. Motivated by this, we study the compilation of CNFs into SDDs by a top-down approach.

This chapter is based on the following contributions. We first identify a subset of SDDs, called Decision-SDDs, which facilitates the top-down compilation of SDDs. We then introduce a top-down algorithm for compiling CNFs into Decision-SDDs, which is harnessed with techniques used in modern SAT solvers, and a new component caching scheme. We finally present empirical results, showing orders-of-magnitude improvement in compilation time, compared to the state-of-the-art, bottom-up SDD compiler.

This chapter is organized as follows. We first introduce the new representation Decision-SDD (Section 6.2). We then provide a formal framework for the compiler (Section 6.3), which is followed by the presentation of the new compilation algorithm (Section 6.4). After presenting experimental results (Section 6.6) we discuss related work (Section 6.7). We close the chapter by presenting the proofs of technical results.

6.2 Decision-SDDs

We will now define the language of Decision-SDD, which is a strict subset of SDDs and a strict superset of OBDDs. Our new top-down compiler will construct Decision-SDDs.

To define Decision-SDDs, we first need to distinguish between internal vtree nodes as follows. An internal vtree node that is not Shannon is called a decomposition node. For instance, vtree node $v = 2$ in Figure 6.1(a) is a decomposition node. An SDD node that is normalized for a Shannon (resp., decomposition) vtree node is called a Shannon (resp., decomposition) decision node. A Shannon decision node has the form $\{(X, \alpha), (\neg X, \beta)\}$, where $X$ is a Shannon variable.
Figure 6.1: A vtree and a respecting Decision-SDD.

Definition 22 (Decision-SDD). A **Decision-SDD** is an SDD in which each decomposition decision node has the form \(\{(p, s_1), (\neg p, s_2)\}\) where \(s_1 = \top\), \(s_1 = \bot\), or \(s_1 = \neg s_2\).

Figure 6.1 shows a Decision-SDD and a corresponding vtree for the CNF \(\{Y \lor \neg Z, \neg X \lor Z, X \lor \neg Y, X \lor Q\}\). The language of Decision-SDDs is complete: every Boolean function can be represented by a Decision-SDD using an appropriate vtree.

For further insights into Decision-SDDs, note that a decomposition decision node must have the form \(\{(f, g), (\neg f, \bot)\}\), \(\{(f, \top), (\neg f, g)\}\), or \(\{(f, \neg g), (\neg f, g)\}\). Moreover, these forms represent the Boolean functions \(f \land g\), \(f \lor g\), and \(f \oplus g\), respectively, where \(f\) and \(g\) are over disjoint sets of variables.

If an SDD is based on a general vtree, it may or may not be a Decision-SDD. However, decision vtree (identified in Chapter 4) guarantees a Decision-SDD.

**Proposition 2.** Let \(v\) be a decision vtree for CNF \(\Delta\). An SDD for \(\Delta\) that respects vtree \(v\) must be a Decision-SDD.
As such, the input to our compiler will be a CNF and a corresponding decision vtree, and the result will be a Decision-SDD for the CNF. Recall that one can always construct a decision vtree for any CNF (as mentioned in Chapter 4).

When every internal vtree node is a Shannon vtree node (i.e., the vtree is right-linear), the Decision-SDD corresponds to an OBDD. A quasipolynomial separation between SDDs and OBDDs was given by [Raz14]. As it turns out, the SDDs used to show this separation are actually Decision-SDDs. We now complement this result by showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size.

**Theorem 15.** Every Decision-SDD with $n$ variables and size $N$ has an equivalent OBDD with size $\leq N^{1+\log n}$.

[XCD12] have identified a class of Boolean functions $f_i$, with corresponding variable orders $\pi_i$, such that the OBDDs based on orders $\pi_i$ have exponential size, yet the SDDs based on vtrees that dissect orders $\pi_i$ have linear size.\(^1\) Interestingly, the SDDs used in this result turn out to be Decision-SDDs as well. Hence, a variable order that blows up an OBDD can sometimes be dissected to obtain a vtree that leads to a compact Decision-SDD. This reveals the practical significance of Decision-SDDs despite the quasipolynomial simulation of Theorem 15.

We will next provide a top-down algorithm for compiling CNFs into Decision-SDDs, which is based on state-of-the-art techniques from SAT solving. Our intention is to provide a formal description of the algorithm, which is precise and detailed enough to be reproducible by the community. We will start by providing a formal description of our framework in Section 6.3, and then present our algorithm in Section 6.4.

\(^1\)A vtree dissects a variable order if the order is generated by a left-right traversal of the vtree.
6.3 A Formal Framework for the Compiler

Modern SAT solvers utilize two powerful and complementary techniques: unit resolution and clause learning. Unit resolution is an efficient, but incomplete, inference rule which identifies some of the literals implied by a CNF. Clause learning is a process which identifies clauses that are implied by a CNF, then adds them to the CNF so as to empower unit resolution (i.e., allows it to derive more literals). These clauses, also called asserting clauses, are learned when unit resolution detects a contradiction in the given CNF. We will neither justify asserting clauses, nor delve into the details of computing them, since these clauses have been well justified and extensively studied in the SAT literature (see, e.g., [MMZ01]). We will, however, employ asserting clauses in our SDD compiler (we employ first-UIP asserting clauses as implemented by RSat [PD07]).

As a first step towards introducing our compiler, we present in Algorithm 5 a modern SAT solver that is based on unit resolution and clause learning (also known as a conflict-driven clause learning (CDCL) solver). This algorithm takes as input a CNF $\Delta$. It maintains a set of clauses $\Gamma$ (for learned clauses) and a decision sequence $D$ (for literal assignments), both of which are initially empty. Given $\Delta, \Gamma,$ and $D$, the solver repeatedly performs the following process. A literal $\ell$ is chosen and added to the decision sequence $D$ (we say that $\ell$ has been decided at level $|D|$). After deciding the literal $\ell$, unit resolution is performed on $\Delta \land \Gamma \land D$. If no contradiction is found, another literal is decided. Otherwise, an asserting clause $\alpha$ is identified. A number of decisions are then erased until we reach the decision level corresponding to the assertion level of clause $\alpha$, at which point $\alpha$ is added to $\Gamma$.

The solver terminates under one of two conditions: either a contradiction is found under an empty decision sequence $D$ (Line 5), or all literals are successfully decided (Line 12). In the first case, the input CNF must be unsatisfiable. In the

\[\text{The assertion level is computed when the clause is learned. It equals the lowest decision level at which unit resolution is guaranteed to derive a new literal using the learned clause.}\]
Algorithm 5: SAT(Δ)

Input: Δ : a CNF

Output: ⊤ if Δ is satisfiable; ⊥ otherwise

1. Γ ← {} // learned clauses
2. D ← ⟨⟩ // decision sequence
3. while true do
   4. if unit resolution detects a contradiction in Δ ∧ Γ ∧ D then
      5. if D = ⟨⟩ then return ⊥
      6. α ← asserting clause for (Δ, Γ, D)
      7. m ← the assertion level of α
      8. D ← the first m decisions of D
      9. Γ ← Γ ∪ {α} // learning clause α
   else
      11. if ℓ is a literal where neither ℓ nor ¬ℓ are implied by unit resolution from Δ ∧ Γ ∧ D then D ← D; ℓ
      else return ⊤

second case, the CNF is satisfiable with D as a (partial) satisfying assignment.³

Algorithm 5 is iterative. Our SDD compiler, however, will be recursive. To further prepare for this recursive algorithm, we will take two extra steps. The first step is to abstract the primitives used in SAT solvers (depicted in Figure 6.2), viewing them as operations on what we shall call a SAT state.

Definition 23 (SAT State). A SAT state is a tuple S = (Δ, Γ, D, I) where Δ and Γ are sets of clauses, D is a sequence of literals, and I is a set of literals, such that Δ ⊨ Γ and Δ ∧ Γ ∧ D ⊢ I. The number of literals in D is called the decision level of S. Moreover, S is said to be satisfiable iff Δ ∧ D is satisfiable.⁴

Here, Δ is the input CNF, Γ is the set of learned clauses, and D is the decision level.

³When D is partial, the variables not appearing in D can take any value assignment.
⁴Without loss of generality, Δ has no empty or unit clauses.
The second step towards presenting our compilation algorithm is a recursive algorithm for counting the models of a CNF, which utilizes the above abstractions (i.e., the SAT state and its associated primitives in Figure 6.2). To simplify the presentation, we will assume an order \( \pi \) of the CNF variables. If \( X \) is the first variable in order \( \pi \), then one recursively counts the models of \( \Delta \land X \), recursively counts the models of \( \Delta \land \neg X \), and then add these results to obtain the model count of \( \Delta \). This is given in Algorithm 6, which is called initially with the SAT state \((\Delta, \emptyset, \langle \rangle, \emptyset)\) to count the models of \( \Delta \). What makes this algorithm additionally
useful for our presentation purposes is that it is exhaustive in nature. That is, when considering variable X, it must process both its phases, X and ¬X. This is similar to our SDD compilation algorithm — but in contrast to SAT solvers which only consider one phase of the variable. Moreover, Algorithm 6 employs the primitives of Figure 6.2 in the same way that our SDD compiler will employ them later.

The following is a key observation about Algorithm 6 (and the SDD compilation algorithm). When a recursive call returns a learned clause, instead of a model count, this only means that while counting the models of the CNF ∆ ∧ D targeted by the call, unit resolution has discovered an opportunity to learn a clause (and learned one). Hence, we must backtrack to the assertion level, add the clause, and then try again (Lines 10 and 19). In particular, returning a learned clause does not necessarily mean that the CNF targeted by the recursive call is unsatisfiable. The only exception is the root call, for which the return of a learned clause implies an unsatisfiable CNF (and, hence, a zero model count) since the learned clause must be empty in this case.\footnote{When the decision sequence D is empty, and unit resolution detects a contradiction in ∆ ∧ Γ, the only learned clause is the empty clause, which implies that ∆ is unsatisfiable (since ∆ ⊧ Γ).}

\section{A Top-Down SDD Compiler}

We are now ready to present our SDD compilation algorithm, whose overall structure is similar to Algorithm 6, but with a few exceptions. First, the SDD compilation algorithm is driven by a vtree instead of a variable order. Second, it uses the vtree structure to identify disconnected CNF components and compiles these components independently. Third, it employs a component caching scheme to avoid compiling the same CNF component multiple times. Note that our algorithm can also be seen as an augmented version of Algorithm 2 (presented in Chapter 4) with powerful SAT solving techniques.
Algorithm 6: \#SAT(\pi, S)

**Input:** \pi: a variable order, S: a SAT state (\Delta, \Gamma, D, I)

**Output:** Model count of \Delta \land D over variables in \pi, or a clause

1. if there is no variable in \pi then return 1
2. \ X \leftarrow \text{first variable in} \ \pi
3. if \ X \text{ or } \neg X \text{ belongs to} \ I \text{ then return} \ \#SAT(\pi \setminus \{ X \}, S)
4. \ h \leftarrow \text{decide literal}(X, S)
5. if \ h \text{ is success then} \ h \leftarrow \#SAT(\pi \setminus \{X\}, S)
6. undo_decide literal(X, S)
7. if \ h \text{ is a learned clause then}
   8. if at_assertion_level(h, S) then
      9. \ h \leftarrow \text{assert clause}(h, S)
     10. if \ h \text{ is success then return} \ #SAT(\pi, S)
     11. else return \ h
   12. else return \ h
8. \ l \leftarrow \text{decide literal}(\neg X, S)
9. if \ l \text{ is success then} \ l \leftarrow \#SAT(\pi \setminus \{X\}, S)
10. undo_decide literal(\neg X, S)
11. if \ l \text{ is a learned clause then}
   12. if at_assertion_level(l, S) then
      13. \ l \leftarrow \text{assert clause}(l, S)
     14. if \ l \text{ is success then return} \ #SAT(\pi, S)
     15. else return \ l
   16. else return \ l
17. return \ h + \ l

Algorithm 7 presents the new SDD compilation method, which is initially called with the SAT state \( S = (\Delta, \emptyset, \langle \rangle, \emptyset) \) and a decision vtree \( v \) for \( \Delta \), to
compile an SDD for CNF $\Delta$. When the algorithm is applied to a Shannon vtrees node, its behavior is similar to Algorithm 6 (Lines 15–44). That is, it basically uses the Shannon variable $X$ and considers its two phases, $X$ and $\neg X$. However, when applied to a decomposition vtrees node $v$ (Lines 5–14), one is guaranteed that the CNF associated with $v$ is decomposed into two components, one associated with the left child $v^l$ and another with the right child $v^r$ (since $v$ is a decision vtrees for $\Delta$). In this case, the algorithm compiles each component independently and combines the results.

We will next show the soundness of the algorithm, which requires some additional definitions. Let $\Delta$ be the input CNF. Each vtrees node $v$ is then associated with:

- $CNF(v)$, $ContextC(v)$, and $ContextV(v)$ as defined before.

- $ContextL(v,I)$: The literals of $ContextV(v)$ appearing in a given set of literals $I$ (context literals of $v$ given $I$).

We start with the following invariant of Algorithm 7.

**Theorem 16.** Consider a call $c2s(v,S)$ to Algorithm 7 with $S = (.,.,D,I)$. Then, $D \subseteq ContextL(v,I)$ and $ContextL(v,I)$ contains exactly one literal for each variable of $ContextV(v)$.

Hence, when calling vtrees node $v$, all its context variables must be either decided or implied. We can now define the CNF component associated with a vtrees node $v$ at state $S$.

**Definition 24.** The component of vtrees node $v$ and state $S = (.,.,.,I)$ is $CNF(v,S) = CNF(v) \land ContextC(v)\mid_\gamma$, where $\gamma = ContextL(v,I)$.

---

[6] Algorithm 7 assumes that certain negations are freely available (e.g., $\neg p$ on Line 14). One can easily modify the algorithm so it returns both an SDD and its negation, making all such negations freely available. We skip this refinement here for clarity of exposition.
Algorithm 7: c2s(v, S)

unique(α) removes an element from α if its prime is ⊥. It then returns s if α = {(p1, s), (p2, s)} or α = {(⊤, s)}; returns p1 if α = {(p1, ⊤), (p2, ⊥)}; else returns the unique SDD node with elements α.

Input: v : a vtree node, S : a SAT state (Δ, Γ, D, I)
Output: A Decision-SDD or a clause

1. if v is a leaf node then
   X ← variable of v
   if either X or ¬X belongs to I then return X or ¬X whichever belongs to I
   else return ⊤
2. else if v is a decomposition vtree node then
   p ← c2s(vl, S)
   if p is a learned clause then
      clean_cache(vl)
   return p
   s ← c2s(vr, S)
   if s is a learned clause then
      clean_cache(v)
   return s
   return unique({(p, s), (¬p, ⊥)})
3. else key ← Key(v, S)
   if cache(key) ̸= nil then return cache(key)
   X ← Shannon variable of v
   if either X or ¬X belongs to I then
      p ← the literal of X that belongs to I
      s ← c2s(vr, S)
      if s is a learned clause then return s
   return unique({(p, s), (¬p, ⊥)})
   s1 ← decide_literal(X, S)
   if s1 is success then s1 ← c2s(vr, S)
   undo_decide_literal(X, S)
   if s1 is a learned clause then
      if at_assertion_level(s1, S) then
         s1 ← assert_clause(s1, S)
         if s1 is success then return c2s(v, S)
      else return s1
   else return s1
   s2 ← decide_literal(¬X, S)
   if s2 is success then s2 ← c2s(vr, S)
   undo_decide_literal(¬X, S)
   if s2 is a learned clause then
      if at_assertion_level(s2, S) then
         s2 ← assert_clause(s2, S)
         if s2 is success then return c2s(v, S)
      else return s2
   else return s2
   α ← unique({(X, s1), (¬X, s2)})
   cache(key) ← α
   return α
Hence, the component $CNF(v, S)$ will only mention variables in vtree $v$. Moreover, the root component ($CNF(v, S)$ with $v$ being the root vtree node) is equal to $\Delta$.

Following is the soundness result assuming no component caching (i.e., while omitting Lines 8, 12, 16, 17 and 43).

**Theorem 17.** A call $c2s(v, S)$ with a satisfiable state $S$ will return either an SDD for component $CNF(v, S)$ or a learned clause. Moreover, if $v$ is the root vtree node, then it will return an SDD for $CNF(v, S)$.

**Theorem 18.** A call $c2s(v, S)$ with an unsatisfiable state $S$ will return a learned clause, or one of its ancestral calls $c2s(v', \cdot)$ will return a learned clause, where $v'$ is a decomposition vtree node.

We now have our soundness result (without caching).

**Corollary 2.** If $v$ is the root vtree node, then call $c2s(v, (\Delta, \{\}, \{\}, \{\}))$ returns an SDD for $\Delta$ if $\Delta$ is satisfiable, and returns an empty clause if $\Delta$ is unsatisfiable.

We are now ready to discuss the soundness of our caching scheme (Lines 8, 12, 16, 17 and 43). This requires an explanation of the difference in behavior between satisfiable and unsatisfiable states (based on Theorem 1 of [SBB04]). Consider the component CNFs $\Delta_X$ and $\Delta_Y$ over disjoint variables $X$ and $Y$, and let $\Gamma$ be another CNF such that $\Delta_X \land \Delta_Y \models \Gamma$ (think of $\Gamma$ as some learned clauses). Suppose that $I_X$ is the set of literals over variables $X$ implied by unit resolution from $\Delta_X \land \Gamma$. One would expect that $\Delta_X \equiv \Delta_X \land I_X$ (and similarly for $\Delta_Y$). Moreover, one would prefer to compile $\Delta_X \land I_X$ instead of $\Delta_X$ as the former can make unit resolution more complete, leading to a more efficient compilation. In fact, this is exactly what Algorithm 7 does, as it includes the learned clauses $\Gamma$ in unit resolution when compiling a component. However, $\Delta_X \equiv \Delta_X \land I_X$ is not guaranteed to hold unless $\Delta_X \land \Delta_Y$ is satisfiable. When this is not the
case, compiling $\Delta_X \land I_X$ will only yield an SDD that implies $\Delta_X$ but is not necessarily equivalent to it. However, this is not problematic for our algorithm, for the following reason. If $\Delta_X \land \Delta_Y$ is unsatisfiable, then either $\Delta_X$ or $\Delta_Y$ is unsatisfiable and, hence, either $\Delta_X \land I_X$ or $\Delta_Y \land I_Y$ will be unsatisfiable, and their conjunction will be unsatisfiable. Hence, even though one of the components was compiled incorrectly, the conjunction remains a valid result. Without component caching, the incorrect compilation will be discarded. However, with component caching, one also needs to ensure that incorrect compilations are not cached (as observed by [SBB04]).

By Theorem 18, if we reach Line 8 or 12, then state $S$ may be unsatisfiable and we can no longer trust the results cached below $v$. Hence, clean_cache$(v)$ on Line 8 and 12 removes all cache entries that are indexed by Key$(v',.)$, where $v'$ is a descendant of $v$. We now discuss Lines 16, 17 and 43.

**Definition 25.** Let $v$ be a vtree node, and let $S = (.,.,.,I)$ and $S' = (.,.,.,I')$ be corresponding SAT states. Let $x \subseteq I$ (resp., $x' \subseteq I'$) be the instantiation of variables appearing in both $v$ and ContextC$(v)$. A function Key$(v,S)$ is called a **component key** iff Key$(v,S) = Key(v,S')$ implies that CNF$(v,S) \land x \equiv CNF(v,S') \land x'$.

Hence, as long as Line 16 uses a component key according to this definition, then caching is sound. The following theorem describes the component key we used in our algorithm.

**Theorem 19.** Consider a vtree node $v$ and a corresponding SAT state $S = (.,.,.,I)$. We define Key$(v,S)$ as the following bit vector: (1) each clause $\delta$ in ContextC$(v)$ is mapped into one bit that captures whether $I \models \delta$, and (2) each variable $X$ that appears in both vtree $v$ and ContextC$(v)$ is mapped into two bits that capture whether $X \in I$, $\neg X \in I$, or neither. Function Key$(v,S)$ is a component key.
6.5 A Top-Down Model Counter

In this section, we will show a new model counter based on Algorithm 7. Indeed, once we compile an SDD using this algorithm, we can perform model counting in time linear in the SDD size. However, this requires us to construct the SDD (and save it in memory), which may create unnecessary overhead in certain cases. Yet, by making simple modifications, we can ensure that Algorithm 7 outputs the model count directly, without the need of constructing the SDD. The following table shows the necessary changes we need to apply.

<table>
<thead>
<tr>
<th>Line</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>return 1</td>
</tr>
<tr>
<td>4</td>
<td>return 2</td>
</tr>
<tr>
<td>14</td>
<td>return left \times right</td>
</tr>
<tr>
<td>23</td>
<td>return s</td>
</tr>
<tr>
<td>42</td>
<td>\alpha \leftarrow (weight(X) \times s_1) + (weight(\neg X) \times s_2)</td>
</tr>
</tbody>
</table>

Table 6.1: Changes to adapt Algorithm 7 to do model counting. \(weight(\ell)\) returns the weight of a literal \(\ell\).

6.5.1 Weighted Model Counting

In many applications (e.g., probabilistic inference), a generalized version of model counting is required, called \textit{weighted model counting} [SBK05, CD08], where the \textit{weighted} model count of a CNF is defined as the sum of the weights of its models — as opposed to only the number of models. The weight of a model is defined as the multiplication of the weights of the literals the model has. So, if each literal has weight 1, then the problem reduces to model counting as we have discussed so far.

It is worth to mention here that Algorithm 7 can also be trivially adapted to
do weighted model counting. All we need is to apply the changes in the following table.

<table>
<thead>
<tr>
<th>Line</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>return weight(X) or weight(\neg X) whichever (X) or (\neg X) belongs to (I)</td>
</tr>
<tr>
<td>4</td>
<td>return weight(X) + weight(\neg X)</td>
</tr>
<tr>
<td>14</td>
<td>return left (\times) right</td>
</tr>
<tr>
<td>23</td>
<td>return weight(X) (\times) s</td>
</tr>
<tr>
<td>42</td>
<td>(\text{sum} \leftarrow (weight(X) \times s_1) + (weight(\neg X) \times s_2))</td>
</tr>
</tbody>
</table>

Table 6.2: Changes to adapt Algorithm 7 to do weighted model counting. weight(\(\ell\)) returns the weight of a literal \(\ell\).

### 6.6 Experiments

#### 6.6.1 Top-Down Compiler

We now present an empirical evaluation of the new top-down compiler. In our experiments, we used two sets of benchmarks. First, we used some CNFs from the iscas85, iscas89, and LGSynth89 suites, which correspond to sequential and combinatorial circuits used in the CAD community. We also used some CNFs available at [http://www.cril.univ-artois.fr/PMC/PMC.html](http://www.cril.univ-artois.fr/PMC/PMC.html), which arise from different applications, such as planning and product configuration. We compiled those CNFs into SDDs and Decision-SDDs. To compile SDDs, we used the SDD package\(^7\). To compile Decision-SDDs, we used our implementation of Algorithm 7.\(^8\) All experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU under 1 hour of time limit and with access to 50GB RAM. We next explain our results shown in Table 6.3.

\(^7\)Available at [http://reasoning.cs.ucla.edu/sdd](http://reasoning.cs.ucla.edu/sdd).

\(^8\)Available at [http://reasoning.cs.ucla.edu/minic2d](http://reasoning.cs.ucla.edu/minic2d).
<table>
<thead>
<tr>
<th>CNF</th>
<th>Without post-processing</th>
<th>With post-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compilation time</td>
<td>SDD size</td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>BU+</td>
</tr>
<tr>
<td>c1355</td>
<td>3423.95</td>
<td>189.0</td>
</tr>
<tr>
<td>c432</td>
<td>1.59</td>
<td>0.14</td>
</tr>
<tr>
<td>c499</td>
<td>1360.05</td>
<td>31.48</td>
</tr>
<tr>
<td>c580</td>
<td>3572.87</td>
<td>896.47</td>
</tr>
<tr>
<td>s1196</td>
<td>764.39</td>
<td>1.86</td>
</tr>
<tr>
<td>s1238</td>
<td>1039.39</td>
<td>2.19</td>
</tr>
<tr>
<td>s1423</td>
<td>1860.56</td>
<td>5.67</td>
</tr>
<tr>
<td>s1488</td>
<td>564.25</td>
<td>0.57</td>
</tr>
<tr>
<td>s1510</td>
<td>2072.46</td>
<td>0.50</td>
</tr>
<tr>
<td>s510</td>
<td>49.02</td>
<td>0.09</td>
</tr>
<tr>
<td>s614</td>
<td>3.84</td>
<td>0.28</td>
</tr>
<tr>
<td>s713</td>
<td>4.08</td>
<td>0.36</td>
</tr>
<tr>
<td>s812</td>
<td>21.56</td>
<td>0.34</td>
</tr>
<tr>
<td>s874</td>
<td>7.71</td>
<td>0.15</td>
</tr>
<tr>
<td>s953</td>
<td>1.92</td>
<td>0.0</td>
</tr>
<tr>
<td>9symml</td>
<td>6.15</td>
<td>0.08</td>
</tr>
<tr>
<td>ahu2</td>
<td>1164.19</td>
<td>0.13</td>
</tr>
<tr>
<td>ahu4</td>
<td>0.71</td>
<td>0.14</td>
</tr>
<tr>
<td>apex0</td>
<td>—</td>
<td>235.06</td>
</tr>
<tr>
<td>frg1</td>
<td>165.61</td>
<td>0.46</td>
</tr>
<tr>
<td>frg2</td>
<td>1876.64</td>
<td>49.76</td>
</tr>
<tr>
<td>term1</td>
<td>537.52</td>
<td>25.36</td>
</tr>
<tr>
<td>tt2</td>
<td>20.48</td>
<td>0.69</td>
</tr>
<tr>
<td>vda</td>
<td>0.14</td>
<td>0.0</td>
</tr>
<tr>
<td>x4</td>
<td>21.22</td>
<td>0.36</td>
</tr>
</tbody>
</table>

Table 6.3: Bottom-up and top-down SDD compilations over *iscas85, iscas89, LGSynth89*, and some sampled benchmarks. BU refers to bottom-up compilation without dynamic minimization and BU+ with dynamic minimization. TD refers to top-down compilation, and TD+ with a single minimization step applied at the end.
The first experiment compares the top-down compiler against the bottom-up SDD compiler. Here, we first generate a decision vtree\(^9\) for the input CNF, and then compile the CNF into an SDD using (1) the bottom-up compiler without dynamic minimization (denoted BU), (2) the bottom-up compiler with dynamic minimization (denoted BU+), and (3) the top-down compiler (denoted TD), using the same vtree. Note that BU+ uses a minimization method, which dynamically searches for better vtrees during the compilation process, leading to general SDDs, whereas both BU and TD do not modify the input decision vtree, hence generating Decision-SDDs with the same sizes. We report the corresponding compilation times and sizes in Columns 2–4 and 6–7, respectively. The top-down Decision-SDD compiler was consistently faster than the bottom-up SDD compiler, regardless of the use of dynamic minimization. In fact, in Column 5 we report the speed-ups obtained by using the top-down compiler against the best result of the bottom-up compiler (i.e., either BU or BU+, whichever was faster). There are 40 cases (out of 61) where we observe at least an order-of-magnitude improvement in time. Also, there are 15 cases where top-down compilation succeeded and both bottom-up compilations failed. However, the situation is different for the sizes, when the bottom-up SDD compiler employs dynamic minimization. In almost all of those cases, BU+ constructed smaller representations. As reported in Column 8, which shows the relative sizes of SDDs generated by TD and BU+, there are 21 cases where BU+ produced an order-of-magnitude smaller SDDs. This is not a surprising result though, given that BU+ produces general SDDs and our top-down compiler produces Decision-SDDs, and that SDDs are a strict superset of Decision-SDDs.

Since Decision-SDDs are a subset of SDDs, any minimization algorithm designed for SDDs can also be applied to Decision-SDDs. In this case, however, the results may not be necessarily Decision-SDDs, but general SDDs. In our sec-

\(^9\)We obtained decision vtrees as explained in Chapter 5.
ond experiment, we applied the minimization method provided by the bottom-up SDD compiler to our compiled Decision-SDDs (as a post-processing step). We then added the top-down compilation times to the post-processing minimization times and reported those in Column 9, with the resulting SDD sizes in Column 10. As is clear, the post-processing minimization step significantly reduces the sizes of SDDs generated by our top-down compiler. In fact, the sizes are almost equal to the sizes generated by BU+ (Column 7). The top-down compiler gets slower due to the cost of the post-processing minimization step, but its total time still dominates the bottom-up compiler. Indeed, it can still be an order-of-magnitude faster than the bottom-up compiler (18 cases). This shows that one can also use Decision-SDDs as a representation that facilitates the compilation of CNFs into general SDDs.

### 6.6.2 Top-Down Model Counter

We now present an empirical evaluation of our new model counter miniC2D. In our experiments, we used 1392 CNF benchmarks available at [http://www.cril.univ-artois.fr/PMC/pmc.html](http://www.cril.univ-artois.fr/PMC/pmc.html), which come from different applications, such as planning and product configuration. We compared our model counter against three different systems: cachet\textsuperscript{10}, c2d\textsuperscript{11}, sharpSAT\textsuperscript{12,13}. All experiments were performed on a 2.6GHz Intel Xeon X5650 CPU under 1 hour of time limit and with access to 8GB RAM.

Figure 6.3 presents the results using scattered plots. For each system mentioned above, there is a plot that compares the corresponding system with miniC2D. In each plot, the points represent timings for counting the models of a specific CNF.

\textsuperscript{10}Available at [http://www.cs.rochester.edu/users/faculty/kautz/Cachet](http://www.cs.rochester.edu/users/faculty/kautz/Cachet).

\textsuperscript{11}Available at [http://reasoning.cs.ucla.edu/c2d](http://reasoning.cs.ucla.edu/c2d).

\textsuperscript{12,13}Available at [https://sites.google.com/site/marcthurley/sharpsat](https://sites.google.com/site/marcthurley/sharpsat).

\textsuperscript{13}c2d compiles a CNF into a Decision-DNNF, and then performs model counting in time linear in the Decision-DNNF size. cachet and sharpsat both output model counting without constructing an NNF, yet their traces are in Decision-DNNF.
using miniC2D and another model counter. The points above the straight line denote instances where miniC2D performs better. Here are a few observations. miniC2D was able to compute model counts of 1042 CNF instances, whereas c2d, cachet, and sharpSAT were able to compute model counts of 1178, 1027, and 1040 CNF instances, respectively. Further, Table 6.4 shows a comparison of the mentioned systems, by showing the number of CNF instances solved by one system but not by another one. Finally, the average counting times were as follows: miniC2D took 164.27 seconds, c2d took 62.69 seconds, cachet took 116.49 seconds, and sharpSAT took 39.45 seconds.

Accordingly, our new model counter has a performance comparable to other state-of-the-art model counters. This is indeed a surprising result as the trace of our algorithm belongs to Decision-SDD, which is not as succinct as the trace of other model counters (which is in Decision-DNNF).

<table>
<thead>
<tr>
<th></th>
<th>miniC2D</th>
<th>c2d</th>
<th>cachet</th>
<th>sharpSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>miniC2D</td>
<td>X</td>
<td>42</td>
<td>147</td>
<td>133</td>
</tr>
<tr>
<td>c2d</td>
<td>178</td>
<td>X</td>
<td>201</td>
<td>181</td>
</tr>
<tr>
<td>cachet</td>
<td>132</td>
<td>50</td>
<td>X</td>
<td>12</td>
</tr>
<tr>
<td>sharpSAT</td>
<td>131</td>
<td>43</td>
<td>25</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 6.4: Comparison among model counters. An entry \((x, y)\) shows the number of instances solved by the system \(x\) but not by the system \(y\).

### 6.7 Related Work

Our algorithm for compiling CNFs into SDDs is based on Algorithm 2, discussed in Chapter 4 and published in [OD14b]. The latter algorithm was proposed to improve a size upper bound on SDDs. However, it did not identify Decision-SDDs, nor did it suggest a practical implementation. The current work makes the
Figure 6.3: Comparison of miniC2D against state-of-the-art model counters.
previously introduced algorithm practical by adding powerful techniques from the
SAT literature and defining a practical caching scheme, resulting in an efficient
compiler that advances the state-of-the-art.

Combining clause learning and component caching was already used in the
context of knowledge compilation [Dar04] and model counting [SBB04]. Yet,
neither of these works described the corresponding algorithms and their prop-
erties at the level of detail and precision that we did here. A key difference be-
tween the presented top-down compiler and the one introduced in [Dar04], called
c2d, is that we compile CNFs into Decision-SDDs, while c2d compiles CNFs into
Decision-DNNFs. These two languages differ in their succinctness and tractabil-
ity (Decision-SDDs we compiled are a strict subset of Decision-DNNFs, and are
less succinct but more tractable). For example, SDDs can be negated in linear
time. Hence, the CNF-to-SDD compiler we introduced can easily be used as a
DNF-to-SDD compiler. For that, we first negate the DNF into a CNF by flipping
the literals and treating each term as a clause. After compiling the resulting CNF
into an SDD, we can negate the resulting SDD efficiently, which would become
the SDD for the given DNF. Since no efficient negation algorithm is known for
d-DNNFs, one cannot use c2d when the original knowledge base is represented
in DNF. We note that we did not evaluate our compiler for compiling DNFS into
SDDs, so we do not know how practical it would be. Still, it can be immediately
used to compile DNFS, which has not been discussed before in the context of
top-down compilation. Another top-down compiler, called eadt, was presented
recently [KLM13], which compiles CNFs into a tractable language that makes
use of decision trees with xor nodes. A detailed comparison of bottom-up and
top-down compilation has been made before in the context of compiling CNFs
into OBDDs [HD04]. Our work can be seen as making a similar comparison for
compiling CNFs into SDDs.
6.A Quasipolynomial Simulation of Decision-SDDs

In this section we will prove Theorem 15, showing that Decision-SDDs can be simulated by OBDDs with at most a quasipolynomial increase in size. We first start with defining a special type of vtree that will be used in the simulation.

Definition 26 (Right-leaning vtree). A vtree is called right-leaning when the number of variables in any right subtree is greater than or equal to the number of variables in the sibling left subtree. In a right-leaning vtree, the left child of a decomposition vtree node is called a light node and the right child is called a heavy node. The number of light nodes on a path from the vtree root to one of its leaves is called the light-length of the path. The light-height of the vtree is the maximum light-length attained by any of its paths.

The following is a key property of right-leaning vtrees that will be used later.

Proposition 3. Let v be a right-leaning vtree over n variables. The light-height of vtree v is at most \( \log n \).

Proof. Let \( v_1, \ldots, v_k \) be the light nodes on the path from the vtree root to a leaf, where \( v_1 \) is the closest to the root and \( v_k \) is the furthest. Node \( v_k \) cannot be a leaf, otherwise its parent would be a Shannon node which is a contradiction with \( v_k \) being light. Hence, the vtree rooted at \( v_k \) must have \( \geq 2 \) variables. Then, the vtree rooted at \( v_k \)'s sibling must have \( \geq 2 \) variables since the vtree is right-leaning. Thus, the vtree rooted at \( v_k \)'s parent must have \( \geq 4 \) variables. Similarly, since \( v_{k-1} \) is an ancestor of \( v_k \), the vtree rooted at \( v_{k-1} \) must have \( \geq 4 \) variables, and the vtree rooted at its parent must have \( \geq 8 \) variables. Continuing to count this way (by induction), vtree rooted at \( v_1 \) must have \( \geq 2^k \) variables. Hence, \( 2^k \leq n \), which implies that \( k \leq \log n \).
We next show that one can always efficiently construct a Decision-SDD that respects a right-leaning vtree from one that is not based on a right-leaning vtree. This will ensure that we can always use Decision-SDDs based on right-leaning vtrees instead of arbitrary ones.

**Theorem 20.** For every Decision-SDD $\alpha$ of size $N$, there is another, equivalent Decision-SDD $\beta$ that respects a right-leaning vtree and has size $p(N)$ for some polynomial $p$. Moreover, $\beta$ can be obtained from $\alpha$ in time $O(N)$.

**Proof.** Let $T$ be the vtree that $\alpha$ respects. We can obtain $\beta$ by converting vtree $T$ into a right-leaning one as follows. For each decomposition vtree node $v$, if the number of variables in the right child is less than those in the left child, we swap the children of vtree node $v$ and adjust the SDD nodes respecting $v$. As $v$ is a decomposition vtree node, we only need to adjust SDD nodes of the form $\{(f, g), (\neg f, \bot)\}$, $\{(f, \top), (\neg f, g)\}$, and $\{(f, \neg g), (\neg f, g)\}$. The adjustment of these nodes leads to $\{(g, f), (\neg g, \bot)\}$, $\{(g, \top), (\neg g, f)\}$, and $\{(g, \neg f), (\neg g, f)\}$, respectively. This can be done in constant time for each SDD node if the negations of SDD nodes in $\alpha$ are available. This can be ensured by a linear-time preprocessing step that goes over the SDD nodes in $\alpha$ once and negates the nodes along the way. Thus, the whole process to obtain $\beta$ takes time linear in $N$ and the size of $\beta$ is $p(N)$ for some polynomial $p$. $\blacksquare$

We now prove Theorem 15, which is restated next (with a simple modification that puts right-leaning vtrees into play).

**Theorem 15.** Let $\alpha$ be a Decision-SDD of size $N$ that respects a right-leaning vtree $v$ over $n$ variables. Then there exists an equivalent OBDD of size $\leq N^{1+\log n}$ that respects the variable order underlying $v$.

**Proof.** We will appeal to the construction in [BLR14], which transforms a DLDD (decomposable logic decision diagrams) into an FBDD, with at most a quasipolynomial increase in size. We first transform SDD $\alpha$ into a DLDD $\beta$ by transforming
the Shannon decision nodes of $\alpha$ into DLDD decision nodes (labeled with Shannon variables), and transforming the decomposition nodes of $\alpha$ into DLDD function nodes (labeled with the functions they represent). Given the correspondence between SDD and DLDD nodes, it is meaningful to talk about the vtree node respected by a DLDD node. It is also meaningful to refer to some DLDD nodes as light or heavy, depending on the vtree node they respect. We will then use the construction of [BLR14] to convert $\beta$ into an FBDD $\gamma$, which is then guaranteed to satisfy the following properties:

1. The size of FBDD $\gamma$ is $\leq N^{1+L} \leq N^{1+\log n}$, where $L$ is the light-height of vtree $v$.

2. The variable ordering on every path from the root of FBDD $\gamma$ to one of its leaves is consistent with the left-right variable ordering of vtree $v$. Hence, FBDD $\gamma$ is actually an OBDD.

The first property holds because on any path from the root of DLDD $\beta$ to one of its leaves, the number of light nodes is bounded by the light-height of vtree $v$, and the light nodes we consider here correspond to the “light edges” used in [BLR14]. The second property holds because the construction of [BLR14] will end up stacking light nodes over their sibling heavy nodes. In other words, assume there is a DLDD node in $\beta$ that respects a decomposition vtree node $u$. Note that its light node $l$ respects some vtree node in $u^l$ and its heavy node $h$ respects some vtree node in $u^r$. The construction is a recursive one. So, suppose that we already processed nodes $l$ and $h$, and they are actually OBDDs that respect the variable order underlying $u^l$ and $u^r$, respectively. The construction will continue by stacking a private copy of $l$ over $h$, which will result in an OBDD that is consistent with the left-right variable ordering of $u$. That is, at each decomposition vtree node $u$, we will create an OBDD that respects the variable order underlying vtree $u$. Thus, the resulting structure would respect the variable order underlying vtree $v$. ■
6.B  Soundness of the Compilation Algorithm

We will now present the proofs of the theorems that were used in the soundness of Algorithm 7 (i.e., Theorems 16–18). We start by listing some assumptions/observations that will be used in the rest of the section. First, given a CNF ∆, we will write ∆ ⊢ I to mean that I is the set of literals derived from ∆ using unit resolution. Second, S will denote a state (∆, Γ, D, I), where ∆ and Γ are sets of clauses, D is a sequence of literals, and I is a set of literals. 14 Third, each (recursive) call c2s(v, S) of Algorithm 7 will take two inputs v and S such that v is a vtree node belonging to a decision vtree for ∆ and S is a SAT state. 15 The latter implies that ∆ |= Γ and ∆ ∧ Γ ∧ D ⊢ I. This holds due to the following three facts: (1) initial call is made with (∆, {}, ⟨⟩, {}) which is a SAT state as there is no unit or empty clause in ∆; (2) any clause added to Γ is a learned clause, which must be implied by ∆; and (3) literals I is always adjusted before making a call (see SAT primitives in Figure 6.2). Finally, a SAT state S is said to be callable iff unit resolution does not detect a contradiction in ∆ ∧ Γ ∧ D. Indeed, S will be callable for each call c2s(v, S). This is true due to the following two facts: (1) initial SAT state (∆, {}, ⟨⟩, {}) is callable as ∆ has no unit or empty clause; and (2) whenever a new SAT state, which is not callable, is constructed during a call, Algorithm 7 backtracks until the contradiction is resolved. So, it will initiate a call only on callable SAT states. We now prove Theorem 16.

**Theorem 16.** Consider a call c2s(v, S) to Algorithm 7 with S = (.., D, I). Then, D ⊆ ContextL(v, I) and ContextL(v, I) contains exactly one literal for each variable of ContextV(v).

**Proof.** Let γ = ContextL(v, I). We first show D ⊆ γ. For that, we show D ⊆ I and Vars(D) ⊆ ContextV(v). The former is due to ∆ ∧ Γ ∧ D ⊢ I. For the latter,

14We will sometimes abuse notation to use D as a set of literals.
15All calls considered in the proofs are assumed to be legal (i.e., can be generated by executing Algorithm 7).
note that when Algorithm 7 decides on a literal (i.e., Line 24 and Line 33), it
undoes its decision after completing a recursive call on the next line (i.e., Line 26
and Line 35). Thus, when call \( \text{c2s}(v, S) \) is made, literals of \( D \) must come from
recursive calls that are not completed. Indeed, these calls can only be made on the
ancestors of \( v \). So, each literal of \( D \) must be a literal of some context variable of
\( v \). That is, \( \text{Vars}(D) \subseteq \text{ContextV}(v) \). We next show \( \gamma \) contains exactly one literal
for each context variable of \( v \). First, we show \( \gamma \) cannot contain two literals of any
variable. Assume otherwise. Then, since \( \gamma \subseteq I \) and \( \Delta \land \Gamma \land D \vdash I \), unit resolution
detects a contradiction in \( \Delta \land \Gamma \land D \), which is a contradiction. We now show \( \gamma \)
contains a literal for each context variable \( X \) of \( v \). Assume otherwise. Let \( p \) be
the Shannon node whose Shannon variable is \( X \). Then, variable \( X \) is not implied
during the corresponding ancestral call to \( p \). As \( p \) is a Shannon node, Algorithm 7
will not recurse on \( p \)'s right child before ensuring \( X \) is implied (see Lines 21, 25,
and 34). That is, call \( \text{c2s}(v, S) \) cannot happen, which is a contradiction. So, a
literal of \( X \) appears in \( I \). Hence, \( I \) contains a literal for each context variable of
\( v \), and so does \( \gamma \).

We remark that Algorithm 7 is recursive. As such, its execution can be viewed
as constructing a tree whose nodes are labeled with recursive calls \( \text{c2s}(. , .) \), and
whose edges are from a recursive call \( R_1 \) to another \( R_2 \) if \( R_2 \) is called within \( R_1 \).
As some of upcoming proofs will be based on this tree, which we denote by \( T \), we
next present two useful observations.

**Proposition 4.** Consider an internal node \( \text{c2s}(v, S) \) on \( T \). Then, \( v \) is either a
decomposition node or a Shannon node.

**Proof.** As \( \text{c2s}(v, S) \) is an internal node, it must have a child. Then, \( v \) cannot
be a leaf \text{vtree} node, as no recursive call is made on Lines 1–4. So, the proposition
follows.

**Proposition 5.** Consider a leaf node \( \text{c2s}(v, S) \) on \( T \). Then, either \( v \) is a leaf
vtree node or $v$ is a Shannon node and call $c2s(v, S)$ returns a clause on Line 31 or Line 32.

Proof. Node $\text{c2s}(v, S)$ can be a leaf node iff no recursive call happens during call $c2s(v, S)$. Due to Line 6, $v$ cannot be a decomposition node. So, $v$ is either a leaf vtree node or a Shannon node. If $v$ is a Shannon node, then the call can return on one of the following lines: 22, 23, 30, 31, 32, 39, 40, 41, and 44. It is not hard to see that no recursive call happens only when call $c2s(v, S)$ returns a clause on Line 31 or Line 32. ■

To prove Theorem 17 and Theorem 18, we will next present some lemmas.

Lemma 14. Consider a call $c2s(v, S)$. Let $S' = (\Delta, \Gamma', D', I')$ be a callable SAT state that appears during call $c2s(v, S)$. Then, we have $\text{ContextL}(v, I) = \text{ContextL}(v, I')$.

Proof. Note that $\Delta \land \Gamma \land D \vdash I$ and $\Delta \land \Gamma' \land D' \vdash I'$. We first show $\Gamma \subseteq \Gamma'$ and $D \subseteq D'$, which implies that $I \subseteq I'$. The former holds as Algorithm 7 never erases learned clauses and $\Gamma$ is obtained before $\Gamma'$. The latter holds as call $c2s(v, S)$ does not undo any decision made before its initiation. Hence, $I \subseteq I'$. Let $\gamma = \text{ContextL}(v, I)$. Since $\gamma \subseteq I$, $\gamma \subseteq I'$. Also, by Theorem 16, $\gamma$ contains exactly one literal for each context variable of $v$. So, $I'$ cannot contain any other literal than $\gamma$ for context variables of $v$. Otherwise, unit resolution detects contradiction in $\Delta \land \Gamma' \land D'$, which violates $S'$ being callable. So, $\gamma = \text{ContextL}(v, I')$. ■

Corollary 3. Consider a call $c2s(v, S)$. Let $S'$ be a callable SAT state that appears during call $c2s(v, S)$. Then, we have $\text{CNF}(v, S) = \text{CNF}(v, S')$.

Lemma 15. Let $S$ be a SAT state and $v$ be a decomposition node of a decision vtree for $\Delta$. Then, we have $\text{CNF}(v, S) = \text{CNF}(v^l, S) \land \text{CNF}(v^r, S)$.

Proof. Since the input vtree is a decision vtree, there is no clause compatible with $v$. Thus, we have $\text{CNF}(v) = \text{CNF}(v^l) \land \text{CNF}(v^r)$, $\text{ContextC}(v) =$
ContextC(vl) ∧ ContextC(vr) and ContextV(vl) = ContextV(vr), implying that
\( CNF(v, S) = CNF(vl, S) ∧ CNF(vr, S) \).

Lemma 16. Consider a call c2s(v, S) on a decomposition node v. Let S’ be a
callable state that appears during call c2s(v, S). Then, CNF(v’, S) = CNF(v’, S’)
for v’s child v’.

Proof. CNF(v, S) = CNF(vl, S) by Corollary 3. Further, due to Lemma 15,
CNF(v, S) = CNF(vl, S) ∧ CNF(vr, S) and CNF(v, S’) = CNF(vl, S’) ∧ CNF(vr, S’).
This implies CNF(vl, S) = CNF(vl, S’) and CNF(vr, S) = CNF(vr, S’), and
thus the lemma holds.

Lemma 17. Consider a call c2s(v, S). Let \( v_1, \ldots, v_n \) be the decomposition nodes
on the path from the vtree root to v (excluding v) and \( v_i \) the child of \( v_i \) not ap-
pearing on the path. Then, \( \Delta|\gamma \equiv \bigwedge_{i=1}^{n} CNF(v_i', S) \land CNF(v, S) \), where \( \gamma = ContextL(v, I) \).

Proof. Note that \( \Delta \equiv \left( \bigwedge_{i=1}^{n} CNF(v_i') \land ContextC(v_i') \right) \land CNF(v) \land ContextC(v) \land \Sigma \), where \( \Sigma \) is a set of clauses that only mention context variables of v. So, the
following holds:
\[
\Delta|\gamma \equiv \left( \bigwedge_{i=1}^{n} CNF(v_i')|\gamma \land ContextC(v_i')|\gamma \right) \land CNF(v)|\gamma \land ContextC(v)|\gamma \land \Sigma|\gamma
\]  
\[
\equiv \left( \bigwedge_{i=1}^{n} CNF(v_i')|\gamma \land ContextC(v_i')|\gamma \right) \land CNF(v)|\gamma \land ContextC(v)|\gamma \quad (6.1)
\]  
\[
\equiv \left( \bigwedge_{i=1}^{n} CNF(v_i') \land ContextC(v_i')|\gamma \right) \land CNF(v) \land ContextC(v)|\gamma \quad (6.2)
\]  
\[
\equiv \bigwedge_{i=1}^{n} CNF(v_i', S) \land CNF(v, S). \quad (6.3)
\]

We now explain why Equations (6.1)–(6.3) hold. Equation (6.1) holds as \( \Sigma|\gamma \equiv \top \).
To see this, note that \( \gamma \) contains exactly one literal for each context variable of v
(see Theorem 16), so that \( \Sigma|\gamma \equiv \bot \) or \( \Sigma|\gamma \equiv \top \). If \( \Sigma|\gamma \equiv \bot \), then unit resolution
detects a contradiction in \( \Delta \land \Gamma \land D \) (since \( \Delta \land \Gamma \land D \vdash I \) and \( \gamma \subseteq I \)). So, \( \Sigma|\gamma \equiv \top \).
Equation (6.2) holds as $\text{CNF}(v'_i)$ and $\text{CNF}(v)$ does not mention any context variable of $v$. Equation (6.3) holds as $\text{CNF}(v'_i, S) \equiv \text{CNF}(v'_i) \land \text{ContextC}(v'_i)|\gamma$ (since $\text{ContextV}(v'_i) \subseteq \text{ContextV}(v)$).

Lemma 18. Consider a call $c2s(v, S)$. Then, $\Delta \land D \equiv \Delta \land \gamma$ where $\gamma = \text{ContextL}(v, I)$.

Proof. Note that $\Delta \land \Gamma \land D \vdash I$. Then, since $\gamma \subseteq I$, $\Delta \land \Gamma \land D \models \gamma$. Further, since $\Delta \models \Gamma$, $\Delta \land \Gamma \land D \equiv \Delta \land D$. So, $\Delta \land D \models \gamma$. By Theorem 16, $D \subseteq \gamma$. So, $\Delta \land D \equiv \Delta \land \gamma$. ■

Lemma 19. Consider a call $c2s(v, S)$. Then, $S$ is satisfiable iff $\Delta|\gamma$ is satisfiable where $\gamma = \text{ContextL}(v, I)$.

Proof. By Lemma 18, $\Delta \land D \equiv \Delta \land \gamma$. Also, as $\gamma$ is a set of literals, $\Delta \land \gamma \equiv \Delta|\gamma \land \gamma$, and so $\Delta \land D \equiv \Delta|\gamma \land \gamma$. Thus, $\Delta \land D$ is satisfiable iff $\Delta|\gamma$ is satisfiable. So, the lemma holds. ■

Lemma 20. Consider a call $c2s(v, S)$ on a Shannon node $v$ with Shannon variable $X$. Let $S' = (\Delta, \Gamma', D', I')$ be a callable SAT state that appears during call $c2s(v, S)$. If a literal $\ell$ of $X$ appears in $I'$, then $\text{CNF}(v^r, S') \equiv \text{CNF}(v, S)|\ell$.

Proof. Assume a literal $\ell$ of $X$ appears in $I'$. We note that $\text{CNF}(v) \land \text{ContextC}(v) \equiv \text{CNF}(v') \land \text{ContextC}(v') \land \text{ContextC}(v^r) \land \Sigma$, where $\Sigma = \text{ContextC}(v^r) \setminus \text{ContextC}(v')$. Also, $\text{CNF}(v') \equiv \top$ as $v$ is a Shannon node and there is no unit or empty clause. Then, the following holds, where $\gamma = \text{ContextL}(v, I)$:

$$\text{CNF}(v, S)|\ell \equiv (\text{CNF}(v) \land \text{ContextC}(v))|\gamma \ell$$
$$\equiv \text{CNF}(v^r) \land (\text{ContextC}(v^r) \land \Sigma)|\gamma \ell$$
$$\equiv \text{CNF}(v^r) \land \text{ContextC}(v^r)|\gamma \ell$$
$$\equiv \text{CNF}(v^r, S').$$  (6.4)

We now explain why Equations (6.4)–(6.5) hold. Note that $\text{ContextV}(v^r) = \text{ContextV}(v) \cup \{X\}$ and $\gamma = \text{ContextL}(v, I')$ (see Lemma 14). Then, $\text{ContextL}(v^r, I') = \text{ContextL}(v, I)$.
γℓ, and so Equation (6.5) holds. Also, Equation (6.4) holds since Σ|γℓ ≡ ⊤. To see this, note that Σ is defined over context variables of v and γℓ contains exactly one literal for each context variable of v, which implies that Σ|γℓ ≡ ⊥ or Σ|γℓ ≡ ⊤. If Σ|γℓ ≡ ⊥, then unit resolution must detect a contradiction in ∆ ∧ Γ' ∧ D' (since ∆ ∧ Γ' ∧ D' ⊢ I' and γℓ ⊆ I'). However, this contradicts with S' being callable. Hence, Σ|γℓ ≡ ⊤.

\[\blacksquare\]

Lemma 21. Consider a call c2s(v, S) with a satisfiable state S. If a literal ℓ of some variable inside v appears in I, then CNF(v, S) ≡ ℓ ∧ CNF(v, S)|ℓ.

Proof. Assume a literal ℓ of some variable X inside v appears in I. Note that ∆ ∧ Γ ∧ D ⊨ I. So, ∆ ∧ Γ ∧ D |≡ ℓ. Further, since ∆ ⊨ Γ, ∆ ∧ Γ ∧ D ≡ ∆ ∧ D. So, ∆ ∧ D |≡ ℓ. Then, due to Lemma 18, ∆ ∧ γ |≡ ℓ where γ = ContextL(v, I). Since γ cannot contain ℓ, ∆|γ |≡ ℓ. By Lemma 17, ∆|γ ≡ Σ ∧ CNF(v, S) where Σ and CNF(v, S) are decomposable CNFs. Here, CNF(v, S) mentions X but Σ does not. Then, given that ∆|γ is satisfiable (see Lemma 19), we have CNF(v, S) |≡ ℓ, which implies that CNF(v, S) ≡ ℓ ∧ CNF(v, S)|ℓ.

\[\blacksquare\]

Lemma 22. Consider a call c2s(v, S) with a satisfiable state S. Let ℓ be a literal of some variable inside v. If ∆ ∧ D ∧ ℓ is unsatisfiable, then CNF(v, S)|ℓ is unsatisfiable.

Proof. Assume ∆ ∧ D ∧ ℓ is unsatisfiable. Since S is satisfiable, ∆ ∧ D is satisfiable. So, (∆ ∧ D)|ℓ must be unsatisfiable. Then, due to Lemma 18, (∆ ∧ γ)|ℓ is unsatisfiable where γ = ContextL(v, I). Since γ cannot contain ℓ, ∆|γℓ is unsatisfiable. Due to Lemma 17, ∆|γ ≡ Σ ∧ CNF(v, S) where Σ and CNF(v, S) are decomposable. As ∆|γ is satisfiable (see Lemma 19), Σ is satisfiable. Since Σ does not mention any variable inside v, Σ|ℓ ≡ Σ. So, ∆|γℓ ≡ Σ ∧ CNF(v, S)|ℓ, and hence CNF(v, S)|ℓ is unsatisfiable.

\[\blacksquare\]

Lemma 23. Consider a call c2s(v, S) on a leaf node v labeled with variable X. Then, CNF(v, S) is equivalent to one of the following: X, ¬X, or ⊤.
Proof. Note that $\text{CNF}(v, S) = \text{CNF}(v) \land \text{ContextC}(v)|\gamma$ where $\gamma = \text{ContextL}(v, I)$. So, by Theorem 16, $\text{CNF}(v, S)$ must be equivalent to one of $X, \neg X, \top, \text{ or } \bot$. We show it cannot be equivalent to $\bot$. Assume otherwise. Note that $\Delta$ has neither an empty clause nor a unit clause. So, $\text{CNF}(v, S) \equiv \text{ContextC}(v)|\gamma$. Thus, $\text{ContextC}(v)$ must include two clauses $\beta_1$ and $\beta_2$ such that $\beta_1|\gamma = X$ and $\beta_2|\gamma = \neg X$. Note that $\Delta \land \Gamma \land D \vdash I$. Then, since $\gamma \subseteq I$, $\beta_1|\gamma = X$, and $\beta_2|\gamma = \neg X$, unit resolution must detect a contradiction in $\Delta \land \Gamma \land D$. As this is a contradiction, $\text{CNF}(v, S)$ cannot be equivalent to $\bot$. ■

**Lemma 24.** Consider a call $c2s(v, S)$. If $\text{CNF}(v, S)$ is unsatisfiable, then call $c2s(v, S)$ will return a (learned) clause.

**Proof.** Assume $\text{CNF}(v, S)$ is unsatisfiable. We use strong induction on the height of node $c2s(v, S)$ on $T$ to show that call $c2s(v, S)$ returns a clause.

**Basis:** Consider a leaf node $c2s(v, S)$ (i.e., at height 0). Due to Lemma 23, $v$ cannot be a leaf vtrees node. Then, by Proposition 5, call $c2s(v, S)$ must return a clause.

**Inductive step:** As an induction hypothesis (IH), assume that the statement holds for the calls at height less than $k$ where $k \geq 1$. Consider an internal node $c2s(v, S)$ (i.e., at height $k$). By Proposition 4, $v$ is either a decomposition node or a Shannon node.

Assume $v$ is a decomposition node. Then, $\text{CNF}(v, S) = \text{CNF}(v^l, S) \land \text{CNF}(v^r, S)$ due to Lemma 15. Since $\text{CNF}(v^l, S)$ and $\text{CNF}(v^r, S)$ are decomposable, one of them must be unsatisfiable. Let $S^l$ (resp., $S^r$) be the SAT state before the call on Line 6 (resp., Line 10). Due to Lemma 16, $\text{CNF}(v^l, S^l) = \text{CNF}(v^l, S)$ and $\text{CNF}(v^r, S^r) = \text{CNF}(v^r, S)$. Then, by IH, either Line 6 or Line 10 must construct a clause (whichever component CNF is unsatisfiable), and hence call $c2s(v, S)$ returns a clause on either Line 9 or Line 13.

Assume $v$ is a Shannon node with Shannon variable $X$. Since $\text{CNF}(v, S)$ is
unsatisfiable, $CNF(v, S)|\ell$ is unsatisfiable for any literal $\ell$. Suppose a literal $\ell$ of $X$ belongs to $I$. Then, the call on Line 21 is made with SAT state $S' = S$. By Lemma 20, $CNF(v^r, S') \equiv CNF(v, S)|\ell$, and so is unsatisfiable. Thus, by IH, Line 21 returns a clause, so does call $c2s(v, S)$. Suppose no literal $\ell$ of $X$ belongs to $I$. We first show that the condition on Line 27 must be satisfied. This can happen iff Line 24 or Line 25 constructs a clause. Note that if Line 24 does not construct a clause, then the call on Line 25 is made with a SAT state $S' = (., ., ., I')$ where $X \in I'$. By Lemma 20, $CNF(v^r, S') \equiv CNF(v, S)|X$, and so is unsatisfiable. So, by IH, Line 25 returns a clause. That is, the condition on Line 27 must be satisfied. Hence, call $c2s(v, S)$ must return on either Line 30, 31 or 32. As Line 31 and Line 32 both return a clause, it remains to show Line 30 returns a clause. Let $S'$ be the state before the call on Line 30. By Corollary 3, $CNF(v, S) = CNF(v, S')$. So, by IH, Line 30 returns a clause. ■

Lemma 25. A call $c2s(v, S)$ with a satisfiable state $S$ will return either an SDD for component $CNF(v, S)$ or a (learned) clause.

Proof. We use strong induction on the height of node $c2s(v, S)$ on $T$ to show that call $c2s(v, S)$ returns either an SDD for $CNF(v, S)$ or a clause.

Basis: Consider a leaf node $c2s(v, S)$ (i.e., at height 0). By Proposition 5, either $v$ is a leaf vtree node or the call returns a clause. Assume $v$ is a leaf vtree node labeled with variable $X$. Then, by Lemma 23, $CNF(v, S)$ is equivalent to one of $X, \neg X$, or $\top$. So, we simply identify $CNF(v, S)$ on Lines 1–4, and return the corresponding SDD.

Inductive step: As an induction hypothesis (IH), assume that the statement holds for the calls at height less than $k$ where $k \geq 1$. Consider an internal node $c2s(v, S)$ (i.e., at height $k$). By Proposition 4, $v$ is either a decomposition node or a Shannon node.

Assume $v$ is a decomposition node. So, call $c2s(v, S)$ can return on one of
the following lines: 9, 13, or 14. Line 9 and Line 13 return clauses. So, assume the call returns on Line 14. That is, Line 6 and Line 10 do not return clauses. Let $S^l$ (resp., $S^r$) be the SAT state before the call on Line 6 (resp., Line 10). Since $S$ is satisfiable, both $S^l$ and $S^r$ are satisfiable (note that decision sequence $D$ stays the same). Then, by IH, Line 6 and Line 10 must return SDDs for $CNF(v^l, S^l)$ and $CNF(v^r, S^r)$, respectively. Further, due to Lemma 16, $CNF(v^l, S^l) = CNF(v^l, S)$ and $CNF(v^r, S^r) = CNF(v^r, S)$. Then, since $CNF(v, S) = CNF(v^l, S) \land CNF(v^r, S)$ (see Lemma 15), Line 14 returns an SDD for $CNF(v, S)$.

Assume $v$ is a Shannon node with Shannon variable $X$. So, call $c2s(v, S)$ can return on one of the following lines: 22, 23, 30, 31, 32, 39, 40, 41 or 44. Lines 22, 31, 32, 40, 41 return clauses. So, we study the remaining lines in the following:

[23] Here, a literal $\ell$ of $X$ belongs to $I$. Then, by Lemma 20, $CNF(v^r, S) \equiv CNF(v, S)|\ell$. So, by IH, Line 21 returns an SDD for $CNF(v, S)|\ell$. Since $CNF(v, S) \equiv \ell \land CNF(v, S)|\ell$ (see Lemma 21), Line 23 returns an SDD for $CNF(v, S)$.

[30, 39] Let $S' = (.,.,D',.)$ be the SAT state before the call on Line 30. It is easy to see that $D' = D$. Then, $S'$ is satisfiable (as $S$ is satisfiable). Also, by Corollary 3, $CNF(v, S') = CNF(v, S)$. So, by IH, Line 30 returns either an SDD for $CNF(v, S)$ or a clause. We can use the same argument for Line 39.

[44] To reach this line, calls on Line 25 and Line 34 should not construct clauses. Note that the call on Line 25 should be made with a SAT state $S' = (\Delta,.,DX,.)$. We now show $S'$ is satisfiable. Assume otherwise. That is, $\Delta \land D \land X$ is unsatisfiable. Then, by Lemma 22, $CNF(v, S)|X$ is unsatisfiable. Note that $CNF(v^r, S') \equiv CNF(v, S)|X$ by Lemma 20. That is, $CNF(v^r, S')$ is unsatisfiable. Then, by Lemma 24, Line 25 returns a clause, which is a contradiction. So, $S'$ is satisfiable. Then, by IH, Line 25 returns an SDD for $CNF(v^r, S')$, which is equivalent to $CNF(v, S)|X$. Similarly, we can show Line 34 returns an SDD for $CNF(v, S)|\neg X$. So, Line 44 returns an SDD for $CNF(v, S)$. ■
Lemma 26. A call \( c_{2s}(v, S) \) with a satisfiable state \( S \) cannot return on neither Line 31 nor Line 40.

**Proof.** Assume call \( c_{2s}(v, S) \) returns on Line 31. Then, Line 29 must construct a clause. Let \( S' = (\Delta, \Gamma', D', .) \) be the SAT state before the call on Line 29. So, unit resolution must detect a contradiction in \( \Delta \wedge \Gamma' \wedge D' \). As \( \Delta \models \Gamma' \), \( \Delta \wedge \Gamma' \wedge D' \equiv \Delta \wedge D' \). So, \( \Delta \wedge D' \) is unsatisfiable. Yet, it is easy to see that \( D' = D \). That is, \( \Delta \wedge D \) is unsatisfiable, which contradicts with \( S \) being satisfiable. Thus, call \( c_{2s}(v, S) \) cannot return on Line 31. Similarly, we can show that it cannot return on Line 40. ■

Lemma 27. Consider a call \( c_{2s}(v, S) \) with a satisfiable state \( S \). If call \( c_{2s}(v, S) \) returns on either Line 32 or Line 41, then \( D \neq \emptyset \).

**Proof.** Assume call \( c_{2s}(v, S) \) returns on Line 32. So, the condition on Line 28 must fail. That is, the current decision level is strictly greater than the assertion level of the learned clause, and hence \( D \neq \emptyset \). Similarly, we can show \( D \neq \emptyset \) if the call returns on Line 41. ■

Lemma 28. Consider a call \( c_{2s}(v, S) \) with a satisfiable state \( S \). If call \( c_{2s}(v, S) \) returns a (learned) clause, then \( D \neq \emptyset \).

**Proof.** Assume call \( c_{2s}(v, S) \) returns a clause. We use strong induction on the height of node \( c_{2s}(v, S) \) on \( T \) to show that \( D \neq \emptyset \).

**Basis:** Consider a leaf node \( c_{2s}(v, S) \) (i.e., at height 0). By Proposition 5, either \( v \) is a leaf vtree node or the call returns a clause on Line 31 or Line 32. As call \( c_{2s}(v, S) \) returns a clause, \( v \) cannot be a leaf node (see Lines 1–4). Also, by Lemma 26, the call cannot return on Line 31. So, by Lemma 27, \( D \neq \emptyset \).

**Inductive step:** As an induction hypothesis (IH), assume that the statement holds for the calls at height less than \( k \) where \( k \geq 1 \). Consider an internal node \( c_{2s}(v, S) \) (i.e., at height \( k \)). By Proposition 4, \( v \) is either a decomposition node or a Shannon node.
Assume \( v \) is a decomposition node. As call \( \text{c2s}(v, S) \) returns a clause, it must return on either Line 9 or Line 13. Suppose it returns on Line 9. So, Line 6 must construct a clause. So, due to IH, \( D \neq \emptyset \). Similarly, we can show \( D \neq \emptyset \) if the call returns on Line 13.

Assume \( v \) is a Shannon node. As call \( \text{c2s}(v, S) \) returns a clause, it must return on one of the following lines: 22, 30, 31, 32, 39, 40 or 41. By Lemma 26, the call cannot return on Line 31 or Line 40. If the call returns on either Line 32 or Line 41, then \( D \neq \emptyset \) by Lemma 27. For the remaining lines (22, 30, 39), using IH, one can easily see that \( D \neq \emptyset \).

We are now ready to prove Theorem 17 and Theorem 18.

**Theorem 17.** A call \( \text{c2s}(v, S) \) with a satisfiable state \( S \) will return either an SDD for component \( \text{CNF}(v, S) \) or a learned clause. Moreover, if \( v \) is the root vtree node, then a learned clause will not be returned.

**Proof.** Due to Lemma 25 and Lemma 28 (note that \( D = \emptyset \) in the initial SAT state).

**Theorem 18.** A call \( \text{c2s}(v, S) \) with an unsatisfiable state \( S \) will return a learned clause, or one of its ancestral calls \( \text{c2s}(v', S') \) will return a learned clause, where \( v' \) is a decomposition vtree node.

**Proof.** Let \( v_1, \ldots, v_n \) be the decomposition vtree nodes on the path from the vtree root to \( v \) (excluding \( v \)) and \( v'_i \) the child of \( v_i \) not appearing on the path. By Lemma 17 and Lemma 19, \( \bigwedge_{i=1}^{n} \text{CNF}(v'_i, S) \land \text{CNF}(v, S) \) is unsatisfiable. So, one of the (decomposable) components \( \text{CNF}(v'_i, S) \) and \( \text{CNF}(v, S) \) is unsatisfiable. Assume \( \text{CNF}(v, S) \) is unsatisfiable. Then, due to Lemma 24, call \( \text{c2s}(v, S) \) returns a clause. Assume one of \( \text{CNF}(v'_i, S) \) is unsatisfiable. Then, by Lemma 15, \( \text{CNF}(v_i, S) \) is unsatisfiable. Consider the ancestral call \( \text{c2s}(v_i, S_i) \) of \( \text{c2s}(v, S) \). By Corollary 3, \( \text{CNF}(v_i, S_i) = \text{CNF}(v_i, S) \), and hence \( \text{CNF}(v_i, S_i) \) is unsatisfiable. Thus, by Lemma 24, call \( \text{c2s}(v_i, S_i) \) returns a clause.
6.C Computing Cache Key

We now prove Theorem 19, which is restated next.

**Theorem 19.** Consider a vtree node $v$ and a corresponding SAT state $S = (., ., ., I)$. Define $Key(v, S)$ as the following bit vector: (1) each clause $\delta$ in $ContextC(v)$ is mapped into one bit that captures whether $I \models \delta$, and (2) each variable $X$ that appears in both vtree $v$ and $ContextC(v)$ is mapped into two bits that capture whether $X \in I$, $\neg X \in I$, or neither. Then function $Key(v, S)$ is a component key.

**Proof.** Let $S' = (., ., ., I')$ be a callable SAT state such that $Key(v, S) = Key(v, S')$. We show $CNF(v, S) \land x \equiv CNF(v, S') \land x'$, where $x$ (resp., $x'$) is a term of variables appearing in both $v$ and $ContextC(v)$ such that $x \subseteq I$ (resp., $x' \subseteq I'$). Due to (2) in the key definition, $x$ and $x'$ are the same. So, it is enough to show $CNF(v, S)|x \equiv CNF(v, S')|x$. Let $\gamma = ContextL(v, I)$ and $\gamma' = ContextL(v, I')$. By Theorem 16, $\gamma$ and $\gamma'$ must include exactly one literal for each variable in $ContextV(v)$. Then, due to (1) in the key definition, we have $ContextC(v)|\gamma x = ContextC(v)|\gamma' x$. So, $CNF(v, S)|x \equiv CNF(v, S')|x$. ■
CHAPTER 7

On Compiling DNNFs without Determinism

In this chapter, we will turn our attention to DNNF compilation. State-of-the-art knowledge compilers generate deterministic subsets of DNNF, which have been recently shown to be exponentially less succinct than DNNF. Hence, we propose a new method to compile DNNFs without enforcing determinism necessarily. Our approach is based on compiling deterministic DNNFs with the addition of auxiliary variables to the input formula. These variables are then existentially quantified from the deterministic structure in linear time, which would lead to a DNNF that is equivalent to the input formula and not necessarily deterministic. On the theoretical side, we show that the new method could generate exponentially smaller DNNFs than deterministic ones. Further, we show that various existing techniques that introduce auxiliary variables to the input formulas can be employed in our framework. On the practical side, we empirically demonstrate that our new method can significantly advance DNNF compilation on certain benchmarks. The material in this chapter is based on the work published in [OD17b].

7.1 Introduction

Decomposability and determinism are two fundamental properties that underlie many tractable representations in propositional logic. Decomposability is the characteristic property of decomposable negation normal form (DNNF) [Dar01a], and adding determinism to DNNF leads to deterministic DNNF (d-DNNF) [Dar01b], which includes many other representations, such as sentential decision diagrams.
The key property of deterministic subsets of DNNF is their ability to render the query of model counting tractable, which is key to probabilistic reasoning (see, e.g., [Rot96, Dar02, CD08]). On the other hand, decomposability without determinism is also sufficient to ensure the tractability of many interesting queries, such as clausal entailment, existential quantification, and cardinality minimization. Indeed, these queries are enough for various applications, which do not require efficient computation of model counting. For example, constructing DNNFs would suffice to perform required reasoning tasks efficiently for model-based diagnosis (e.g., [Dar00, Bar05, HD05]) and testing (e.g., [SHS10, SS10]).

However, state-of-the-art knowledge compilers all generate deterministic subsets of DNNF (see, e.g., [Dar04, MMB12, OD15]). Yet, perhaps unsurprisingly, the addition of determinism comes with a cost of generating less succinct representations. In particular, as recently shown [BCM16], DNNF is exponentially more succinct than its deterministic subsets. Therefore, for those applications where only decomposability is sufficient, compiling a deterministic subset of DNNF not only implies performing more work than necessary, but it could also result in generating larger DNNFs which would make reasoning tasks less efficient (if compilation is possible at all). Still, all existing compilers that we know of to enforce decomposability also ensure determinism.

In this chapter, we focus on compiling DNNFs without enforcing determinism, and make several contributions on this front. Our main contribution is a new methodology to compile DNNFs by leveraging existing knowledge compilers. The key insight behind our approach is a new type of equivalence relation between two Boolean functions: a Boolean function $f(X)$ over variables $X$ is equivalent modulo forgetting to another Boolean function $g(X, Y)$ over variables $X$ and $Y$ iff existentially quantifying (also known as, forgetting) variables $Y$ from $g$ results in a function equivalent to $f$. The relevance of this notion to DNNF compilation is the
well-known result that one can forget arbitrarily many variables on a given DNNF in linear time in the DNNF size, without losing the property of decomposability but not necessarily determinism [Dar01a]. Thus, instead of compiling function f directly, one can compile function g into a deterministic DNNF using existing compilers, on which forgetting variables Y would result in a DNNF that is not necessarily deterministic and equivalent to f.

The usefulness of our new approach depends on two important questions, which we address in this paper both theoretically and empirically: (i) to what extent does forgetting variables help in obtaining compact DNNFs that are not deterministic compared to ones with determinism, and (ii) how can one identify functions that are equivalent modulo forgetting. On the theoretical side, we present two main results. First, we show that even forgetting a single auxiliary variable can lead to exponential difference between sizes of DNNFs with and without determinism. Second, we study various existing approaches, such as Tseitin transformation [Tse68], extended resolution [Tse70], and bounded variable addition (BVA) [MHB12], where auxiliary variables are introduced to formulas, mostly to obtain an equisatisfiable formula so that the SAT task can be performed or made more efficient.¹ We show that those existing techniques indeed correspond to generating functions that are equivalent modulo forgetting, therefore offering some practical techniques for realizing our approach. In particular, we show that BVA can generate CNFs without increasing the treewidth of the input CNF much in the worst case, and could potentially reduce CNFs with unbounded treewidth to ones with bounded treewidth. Since CNF to d-DNNF compilation is tractable for bounded treewidth [Dar01a], this result shows the potential of BVA for DNNF compilation. On the practical side, we demonstrate that BVA, which turns out to be useful for SAT solving, can significantly advance DNNF compilation.

This chapter is organized as follows. We first describe our new method in detail

¹Two formulas are equisatisfiable if one is satisfiable precisely when the other is.
(Section 7.2). This is followed by showing that forgetting auxiliary variables can lead to exponential separation between DNNFs with and without determinism (Section 7.3). We then provide a treatment of various existing approaches in the literature as equivalent modulo forgetting transformations (Section 7.4). After providing an empirical evaluation of our new approach (Section 7.5), we conclude with a discussion on related work (Section 7.6).

### 7.2 Compiling DNNFs through Forgetting Variables

In this section, we will describe the proposed methodology, which is based on a new type of equivalence relation between two functions.

**Definition 27.** Let $f(X)$ and $g(X, Y)$ be two Boolean functions, where variables $X$ and $Y$ are disjoint. Then function $f$ is said to be **equivalent modulo forgetting** (emf) to function $g$ iff the following holds:

$$f(X) \equiv \exists Y. g(X, Y).$$

Intuitively, the models of functions $f$ and $g$ match on their values over variables $X$. Specifically, for each model $x$ of $f$, there must exist an instantiation $y$ such that $xy$ is a model of $g$. Similarly, for each model $xy$ of $g$, $x$ must be a model of $f$. In other words, function $f$ says everything function $g$ says about variables $X$. Hence, variables $Y$ only act as *auxiliary* from the view of function $f$. We note that the model counts of $f$ and $g$ are not necessarily the same.

We utilize this notion in compiling DNNFs as shown in Algorithm 8. Here, to compile a DNNF representation of a function $f(X)$, we first obtain another function $g(X, Y)$ that is emf to function $f$, with variables $Y$ being auxiliary (Line 1). Clearly, the specific method to construct function $g$ would depend on the input representation of $f$. We will discuss different ways for that later in Section 7.4 when the input is a CNF. Once function $g$ is constructed, we compile a
Algorithm 8: \textit{DNNF}(f)

\textbf{Input:} \( f(X) \) : a Boolean function over variables \( X \)

\textbf{Output:} constructs a DNNF representation of function \( f \)

\begin{algorithmic}
\STATE \( g(X, Y) \leftarrow \text{emf}(f) \)
\STATE \( \Delta \leftarrow \text{compile} \ g(X, Y) \) using a d-DNNF compiler
\STATE \( \Gamma \leftarrow \text{forget variables} \ Y \) from \( \Delta \)
\STATE \textbf{return} \( \Gamma \)
\end{algorithmic}

deterministic DNNF representation of it using an off-the-shelf knowledge compiler (Line 2). Finally, we forget auxiliary variables \( Y \) from the compiled structure (Line 3). This would generate a DNNF representation of the input as \( g \) is emf to function \( f \).

\textbf{Proposition 6.} Algorithm 8 returns a DNNF representation of its input.

We remark that the last step of Algorithm 8 (Line 3) can be performed in time linear in the size of the d-DNNF. This is due to the property of decomposability, which supports linear time multiple-variable forgetting: all one needs is to replace auxiliary literals with the constant \( \top \) in the d-DNNF. The result is guaranteed to remain decomposable (DNNF), but not necessarily deterministic (d-DNNF). In fact, as we show in the next section, this could lead to an exponentially more succinct representation of the function \( f(X) \), which can be thought of as a compensation for losing the ability to perform model counting efficiently on function \( f(X) \). An example of this procedure is depicted in Figure 7.1, where we forget variables \( X, Z \) from a d-DNNF.

\section*{7.3 An Exponential Separation by Forgetting Variables}

In this section, we address the following question: to what extent does forgetting auxiliary variables help in obtaining compact, non-deterministic DNNFs compared
Figure 7.1: Forgetting variables $X, Z$ from a d-DNNF. Decomposability is preserved but determinism is lost.

Do we next state our main result, showing that exponentially more compact representations can be obtained.

**Theorem 21.** There exist two classes of Boolean functions $f_n(X)$ and $g_n(X, Z)$ such that: (i) $f_n$ is emf to $g_n$, (ii) the size of each d-DNNF computing $f_n$ is at least exponential in $n$, and (iii) there is a d-DNNF computing $g_n$ whose size is polynomial in $n$.

In other words, while it is not feasible to compile a d-DNNF representation of function $f_n$, one can obtain a compact DNNF representation of $f_n$ through forgetting a single auxiliary variable from the compact d-DNNF representation of function $g_n$. Note that obtaining a DNNF for function $f_n$ directly is not feasible practically as existing knowledge compilers enforce determinism as well.

We next present the proof of Theorem 21, which uses a class of functions that was used to show an exponential separation between DNNFs and d-DNNFs [BCM16].

Let $M$ be an $n \times n$ matrix of Boolean variables. Let $R_1, \ldots, R_n$ be the rows of $M$ and $C_1, \ldots, C_n$ be the columns of $M$ ($R_i$, $C_i$, and $M$ will be treated as sets...
of variables). Let $h_n$ be the class of functions over $n$ variables evaluating to 1 iff the sum of its inputs is divisible by 3. Consider the following function defined on the variables of $\mathbf{M}$ and variable $Z$:

$$g_n(\mathbf{M}, Z) = (Z \land \text{row}_n(\mathbf{M})) \lor (\neg Z \land \text{col}_n(\mathbf{M})),$$

where $\text{row}_n$ and $\text{col}_n$ are defined by

$$\text{row}_n(\mathbf{M}) = \bigoplus_{i=1}^{n} h_n(\mathbf{R}_i), \quad \text{col}_n(\mathbf{M}) = \bigoplus_{i=1}^{n} h_n(\mathbf{C}_i).$$

Finally, let $f_n$ be the following function defined on the variables of $\mathbf{M}$:

$$f_n(\mathbf{M}) = \text{row}_n(\mathbf{M}) \lor \text{col}_n(\mathbf{M}).$$

Clearly, $f_n(\mathbf{M}) \equiv \exists Z. g_n(\mathbf{M}, Z)$, and hence function $f_n$ is emf to function $g_n$. Indeed, function $f_n$ is the Sauerhoff function [Sau03], which was used in the exponential separation of DNNFs from d-DNNFs [BCM16]. That is, $f_n$ has a polynomial size DNNF representation, but each deterministic DNNF computing it is exponential in size. Finally, since functions $\text{row}_n$ and $\text{col}_n$ both have polynomial size OBDDs (a subset of d-DNNF) [Weg00], function $g_n$ has a polynomial size deterministic DNNF representation. Thus, Theorem 21 holds.

The following corollary shows how this result strengthens existing results for deterministic DNNFs and FBDDs, which were only established under a standard complexity-theoretic assumption (i.e., P $\neq$ NP) [DM02].

**Corollary 4.** d-DNNF and FBDD do not support polytime forgetting of a bounded number of variables, nor do they support polytime bounded disjunction.

Theorem 21 reveals the usefulness of our new approach in theory. To make it useful in practice, we need to identify transformations that would produce emf formulas, which is discussed next.
7.4 EMF Transformations

In this section, we address the following question: how can one identify functions that are equivalent modulo forgetting?

We will study some existing techniques for CNFs that incorporate auxiliary variables, mostly to get an equisatisfiable CNF. For each technique, we will demonstrate that the produced equisatisfiable CNF is indeed emf to the input CNF. We first formally define a notion of transformation that will be used to identify methods producing emf formulas.

**Definition 28.** Let $T$ be an algorithm that takes as input a Boolean function $f(X)$ and outputs another Boolean function $g(X, Y)$, where $X$ and $Y$ are disjoint. Then algorithm $T$ is said to be an emf transformation iff function $f$ is emf to function $g$.

Given this definition, we next present some emf transformations that exist in the literature.

7.4.1 Tseitin Transformation

State-of-the-art SAT solvers require their input to be a Boolean formula in CNF. When this is not the case, one has to first transform the input into a CNF. The naive approach here is to use the famous De Morgan’s law and the distributive property, which preserves logical equivalence. However, this can easily blow-up the CNF size exponentially. Thus, one typically applies Tseitin transformation [Tse68], which converts a Boolean formula into an equisatisfiable CNF by adding auxiliary variables with only a linear increase in size. In fact, Tseitin transformation does more than constructing an equisatisfiable CNF. In particular, it guarantees two more properties [Tse68]:

1. Dropping auxiliary variables from a model of the constructed CNF would yield a model of the input formula;
(2) Any model of the input formula can be extended to be a model of the constructed CNF.

As we prove next, these two properties make Tseitin transformation an emf transformation, as well as any other transformation that satisfies them.

**Theorem 22.** Let $T$ be a transformation that satisfies the two properties above. Then $T$ is an emf transformation.

Let $f(X)$ be the input function to transformation $T$, and let $g(X, Y)$ be the output function of this transformation, where variables $Y$ are introduced during the transformation. We want to show that $f(X) \equiv \exists Y. g(X, Y)$.

Let $x$ be a model of $\exists Y. g(X, Y)$. We will show that $x$ is also a model of $f(X)$. Since $x$ is a model of $\exists Y. g(X, Y)$, there must be an instantiation $y$ such that $xy$ is a model of $g(X, Y)$. Then, by the first property above, $x$ must be a model of $f(X)$.

Let $x$ be a model of $f(X)$. We will show that $x$ is also a model of $\exists Y. g(X, Y)$. Due to the second property above, there must exist an instantiation $y$ such that $xy$ is a model of $g(X, Y)$. In this case, $x$ must be a model of the function $g(X, Y) | y$. Note that one can view $\exists Y. g(X, Y)$ as the function

$$g(X, Y) | y_1 \lor \ldots \lor g(X, Y) | y_k,$$

where $y_1, \ldots, y_k$ are all possible instantiations of variables $Y$. Thus, $x$ must be a model of $\exists Y. g(X, Y)$.

Therefore, Theorem 22 holds, which immediately implies that Tseitin transformation is an emf transformation.

**Proposition 7.** Tseitin transformation is an emf transformation.

Accordingly, we can apply Tseitin transformation to compile DNNF when the input is not in CNF, which is also the required format for most knowledge compilers.
7.4.2 Extended Resolution

Resolution is a powerful rule of inference that has been used in SAT solving [Rob65]. Specifically, iterating the following rule repeatedly in a certain way would tell whether a CNF is satisfiable or not:

\[
\frac{X \lor \alpha \quad \neg X \lor \beta}{\alpha \lor \beta},
\]

where \(X\) is a variable and \(\alpha\) and \(\beta\) are clauses. This rule states that whenever the clauses in the premise appear in a CNF, one can increment the CNF by adding the clause in the conclusion, without changing the logical content of the CNF (i.e., preserving logical equivalence). Here, \(\alpha \lor \beta\) is called the resolvent obtained by resolving variable \(X\) on \(X \lor \alpha\) and \(\neg X \lor \beta\).

It turns out that resolution could generate only exponentially long proofs of unsatisfiability for certain families of formulas (see, e.g., the Pigeonhole principle [Hak85]). To remedy this, extended resolution is introduced, which is a more powerful generalization of resolution that includes an additional rule, called the extension rule [Tse68]. Accordingly, extended resolution allows one to increment the CNF with the addition of clauses of the form \(X \leftrightarrow \ell_1 \lor \ell_2\), where \(X\) is an auxiliary variable that does not appear in the CNF and literals \(\ell_1\) and \(\ell_2\) appear in the CNF. Then one can apply the resolution rule as before. This simple addition creates an exponentially more powerful proof system than resolution, as extended resolution can generate polynomially sized proofs where regular resolution can only generate exponentially sized proofs [Coo76].

Indeed, extended resolution constructs an equisatisfiable CNF, and thus applying resolution on it produces correct results for SAT solving. This technique has also been shown to be useful in practice of SAT solving, where different schemes for

\[\text{More specifically, } X \leftrightarrow \ell_1 \lor \ell_2 \text{ can be replaced with the clauses } \neg X \lor \ell_1 \lor \ell_2, X \lor \neg \ell_1, \text{ and } X \lor \neg \ell_2.\]
applying the extension rule have been suggested [Hua10, AKS10, Man14]. Hence, its usage could potentially be extended to DNNF compilation, given that we will now show it is indeed an emf transformation.

We will now prove the following result, which generalizes extended resolution.

**Theorem 23.** Let \( f(X), \alpha^1(X), \ldots, \alpha^n(X) \) be Boolean functions. Consider the class of Boolean function

\[
g_n(X, Y) = f(X) \land (Y_1 \Leftrightarrow \alpha^1(X)) \land \ldots \land (Y_n \Leftrightarrow \alpha^n(X)),
\]

where \( Y = \{Y_1, \ldots, Y_n\} \) is disjoint from \( X \). Then function \( f \) is emf to function \( g_n \).

We want to show that \( f(X) \equiv \exists Y \cdot g_n(X, Y) \). For that, we will use the following simplification \( n \) times:

\[
\exists Y \cdot g_n \equiv \exists Y_1, \ldots, Y_{n-1} \cdot \exists Y_n \cdot f(X) \land \bigwedge_{i=1}^{n} Y_i \Leftrightarrow \alpha^i(X) \tag{7.1}
\]

\[
\equiv \exists Y_1, \ldots, Y_{n-1} \cdot f(X) \land \left( \bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha^i(X) \right) \land \exists Y_n \cdot Y_n \Leftrightarrow \alpha^n(X) \tag{7.2}
\]

\[
\equiv \exists Y_1, \ldots, Y_{n-1} \cdot f(X) \land \bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha^i(X) \tag{7.3}
\]

\[
\equiv \exists Y_1 \cdot f(X) \land (Y_1 \Leftrightarrow \alpha_1(X))
\equiv f(X).
\]

Equation (7.1) is due to the definition of multiple-variable forgetting. Equation (7.2) holds as \( f(X) \land \bigwedge_{i=1}^{n-1} Y_i \Leftrightarrow \alpha_i(X) \) does not mention variable \( Y_n \). Equation (7.3) holds as forgetting variable \( Y_n \) from \( Y_n \Leftrightarrow \alpha_n(X) \) is equivalent to the trivial function \( \top \).

Assuming that \( f(X) \) is a CNF, replacing each \( \alpha^i(X) \) with a clause of two literals of variables \( X \) would clearly correspond to the extension rule of extended resolution.
Proposition 8. Extended resolution is an emf transformation.

7.4.3 Bounded Variable Addition

Bounded variable addition (BVA) is a preprocessing technique introduced for SAT solving [MHB12]. The goal here is to reduce the sum of the number of variables and clauses of a CNF by introducing auxiliary variables, without losing the ability of answering the SAT query. It is based on resolution as described next.

Let $C_X$ be a set of clauses containing literal $X$ and $C_{\neg X}$ a set of clauses containing literal $\neg X$. Let $C_X \bowtie C_{\neg X}$ denote the set of resolvents one would obtain by resolving $X$ on clauses in $C_X$ and $C_{\neg X}$. Given a CNF $\Delta$ and an auxiliary variable $X$ that does not appear in $\Delta$, BVA looks for sets of clauses $C_X$ and $C_{\neg X}$ such that $C_X \bowtie C_{\neg X}$ belongs to $\Delta$ and $|C_X \bowtie C_{\neg X}| > |C_X| + |C_{\neg X}|$. In this case, BVA replaces clauses $C_X \bowtie C_{\neg X}$ with clauses $C_X$ and $C_{\neg X}$. For instance, consider the following CNF:

$$\Delta = (A \lor D) \land (B \lor D) \land (C \lor D) \land (A \lor E) \land (B \lor E) \land (C \lor E).$$

By adding an auxiliary variable $X$, we can obtain the following CNF which has fewer clauses than $\Delta$:

$$\Sigma = (A \lor \neg X) \land (B \lor \neg X) \land (C \lor \neg X) \land (D \lor X) \land (E \lor X).$$

Indeed, $\Delta$ is equisatisfiable to $\Sigma$, and thus one can feed $\Sigma$ to a SAT solver, instead of $\Delta$.

The authors of [MHB12] also developed a heuristic for applying the BVA transformation on CNFs, which is a greedy algorithm that searches for clause-patterns in the input CNF. This heuristic, which was shown useful in SAT solving, offers a practical direction for realizing our DNNF compilation method due to the following result.

Proposition 9. Bounded variable addition is an emf transformation.
We will now prove the above proposition. In particular, let $\Delta(X)$ be a CNF and $\Sigma(X, Y)$ be the CNF obtained by applying BVA on $\Delta$, where $Y$ is the auxiliary variable added during the process. Then we want to show that $\Delta(X) \equiv \exists Y. \Sigma(X, Y)$.

Let $\Gamma_Y = \bigwedge_{i=1}^{m} Y \lor \alpha_i$ and $\Gamma_{\neg Y} = \bigwedge_{j=1}^{k} \neg Y \lor \beta_j$ be the clauses containing literals $Y$ and $\neg Y$ in CNF $\Sigma$, respectively. Then, due to the BVA process, we can rewrite CNF $\Delta$ as the CNF $\Phi \land \Gamma_Y \bowtie \Gamma_{\neg Y}$ where $\Phi$ is a CNF. Moreover, CNF $\Sigma$ is equivalent to $\Phi \land \Gamma_Y \land \Gamma_{\neg Y}$. In this setting, we have the following equations:

\[
\exists Y. \Sigma(X, Y) \equiv \exists Y. \Phi \land \Gamma_Y \land \Gamma_{\neg Y} \\
\equiv \Phi \land \exists Y. \Gamma_Y \land \Gamma_{\neg Y} \\
\equiv \Phi \land \left( (\Gamma_Y \land \Gamma_{\neg Y}) \land Y \lor (\Gamma_Y \land \Gamma_{\neg Y}) \land \neg Y \right) \\
\equiv \Phi \land \left( \Gamma_{\neg Y} \land Y \lor \Gamma_Y \land \neg Y \right) \\
\equiv \Phi \land \left( \bigwedge_{j=1}^{k} \beta_j \lor \bigwedge_{i=1}^{m} \alpha_i \right) \\
\equiv \Phi \land \left( \bigwedge_{j=1}^{k} \bigwedge_{i=1}^{m} \beta_j \lor \alpha_i \right) \\
\equiv \Phi \land \Gamma_Y \bowtie \Gamma_{\neg Y} \\
\equiv \Delta(X).
\]

**Treewidth and BVA** We now identify another guarantee that comes with the BVA transformation. As discussed in earlier chapters, treewidth is a well-known graph-theoretic property, which has been extensively used as a parameter that renders many hard reasoning tasks tractable when being small. In the context of knowledge compilation, it is known that compiling a CNF into a deterministic DNNF can be done in the worst case in time that is linear in the number of variables and exponential in the treewidth of the CNF primal graph [Dar01a]. Therefore, a CNF with a bounded treewidth can easily be compiled into a deterministic DNNF.
We will next present two results regarding the effects of the BVA transformation on the primal treewidth of CNFs, whose proofs are delegated to the appendix at the end of the chapter. Our first result is the following guarantee.

**Theorem 24.** Let $\Delta$ be a CNF whose primal treewidth is $w$. Let $\Sigma$ be the CNF obtained by applying the BVA transformation $k$ times on CNF $\Delta$. Then the primal treewidth of $\Sigma$ is at most $w + k$.

Hence, the BVA transformation would not affect the treewidth much in the worst case, when applied a small number of times. Moreover, as we present next, the BVA transformation could potentially reduce the treewidth from an unbounded value to a bounded value.

**Theorem 25.** There exists a class of CNFs $\Delta_n$ over $n^3$ variables such that: (i) the primal treewidth of $\Delta_n$ is unbounded (at least $2n$), and (ii) applying the BVA transformation two times on $\Delta_n$ can generate a CNF whose primal treewidth is bounded (at most 2).

Theorem 25 implies that the BVA transformation can generate a CNF whose compilation to deterministic DNNF is easy, whereas this cannot be guaranteed on the input CNF (as the treewidth is unbounded). Therefore, Algorithm 8 can easily compile a DNNF in this case, if the BVA transformation is applied. Yet, there is no guarantee for compiling a deterministic DNNF with existing compilers, without applying the BVA transformation. Indeed, we will confirm this empirically in our experiments, where the following class of CNFs $\Delta_n^a$ will be considered:

$$
\bigwedge_{1 \leq i, j, k \leq n} X_i \lor Y_j \lor Z_k.
$$

This class of CNFs has unbounded treewidth. On the other hand, the following class of CNFs $\Delta_n^b$ can be identified by the BVA transformation, which has bounded treewidth.

$$
\left( \bigwedge_{1 \leq i \leq n} A \lor X_i \right) \land \left( \bigwedge_{1 \leq j \leq n} \neg A \lor B \lor Y_j \right) \land \left( \bigwedge_{1 \leq k \leq n} \neg B \lor Z_k \right).
$$
Note that we added two auxiliary variables $A, B$ into CNF $\Delta^a_n$, and reduced the number of clauses from $n^3$ to $3n$.

### 7.5 Experiments

In this section, we will empirically demonstrate the applicability of Algorithm 8 in compiling DNNFs, when coupled with the BVA transformation. In particular, we compile CNFs into DNNF and deterministic DNNF. For the latter, we use the c2d compiler,\footnote{Available at \url{http://reasoning.cs.ucla.edu/c2d}.} and for the former, we use the same compiler after transforming CNFs using the preprocessor Coprocessor\footnote{Available at \url{http://tools.computational-logic.org/content/riss.php}.} and forgetting auxiliary variables after the compilation.

We evaluated the mentioned systems on two different benchmarks. First, we used the manually constructed class of CNFs $\Delta^a_n$ (described in Section 7.4) for values of $n \in \{10, 15, 30, 50, 75, 100\}$. Second, we used some CNF encodings of wire routing problems in field-programmable gate arrays (FPGA) [NAS04]. The goal here is to decide if a routing configuration is possible. That is, given $m$ connections and $k$ channels on an FPGA (denoted $\text{fpga}_m n$), the satisfiability of the CNF encoding would imply that the routing of $m$ connections through $k$ channels is possible. Our experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU with a 1 hour time limit and a memory limit of 8GB RAM.

Table 7.1 highlights the results on CNFs $\Delta^a_n$. According to this, our approach (c2d\_FORGET) recognizes the tractability of the CNF instances by introducing two auxiliary variables (as shown in Section 7.4), and thus it compiles the instances quickly and compactly. On the other hand, the traditional approach (c2d) performed poorly as it could not finish compilation after $n = 15$.

For the FPGA routing problems, we first present some statistics of the CNF
<table>
<thead>
<tr>
<th>$\Delta_n^a$</th>
<th>c2D_FORGET</th>
<th>c2D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#node</td>
<td>#edge</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>15</td>
<td>57</td>
<td>58</td>
</tr>
<tr>
<td>30</td>
<td>102</td>
<td>103</td>
</tr>
<tr>
<td>50</td>
<td>162</td>
<td>163</td>
</tr>
<tr>
<td>75</td>
<td>237</td>
<td>238</td>
</tr>
<tr>
<td>100</td>
<td>312</td>
<td>313</td>
</tr>
</tbody>
</table>

Table 7.1: Experimental results on CNFs $\Delta_n^a$. c2D_FORGET is our approach for compiling DNNFs without determinism. All timings are in seconds.

instances before and after the preprocessing in Table 7.2. We now highlight the results in Table 7.3. Accordingly, our approach is clearly superior than the traditional approach as we can compile 5 instances which otherwise could not be compiled. In the remaining 2 instances, not only does our approach produce DNNFs faster, but it also constructs more compact representations. Therefore, our approach improves the performance of DNNF compilation on these FPGA problems.

7.6 Related Work

The closest related work to ours is perhaps the work of [PD08], in which the authors identified a subset of DNNF, called structured DNNF. The significance here is that this subset supports a polynomial time conjoin operation [PD08], while general DNNF do not support this (unless $P = NP$) [DM02]. Due to this operation, one can compile CNFs incrementally in a bottom-up fashion into a structured DNNF. That is, after representing each clause as a structured DNNF (which can be done easily), one can conjoin clauses one by one until a structured DNNF is
<table>
<thead>
<tr>
<th>Instance</th>
<th>Before BVA</th>
<th>After BVA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#variable</td>
<td>#clause</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>448</td>
</tr>
<tr>
<td></td>
<td>135</td>
<td>549</td>
</tr>
<tr>
<td></td>
<td>144</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td>162</td>
<td>684</td>
</tr>
<tr>
<td></td>
<td>198</td>
<td>968</td>
</tr>
<tr>
<td></td>
<td>216</td>
<td>1128</td>
</tr>
<tr>
<td></td>
<td>176</td>
<td>759</td>
</tr>
</tbody>
</table>

Table 7.2: Some stats on CNF encodings of FPGA routing problems, before and after preprocessing using the BVA transformation.

<table>
<thead>
<tr>
<th>Instance</th>
<th>C2D_FORGET</th>
<th>C2D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#node</td>
<td>#edge</td>
</tr>
<tr>
<td></td>
<td>38,601</td>
<td>116,399</td>
</tr>
<tr>
<td></td>
<td>37,528</td>
<td>107,316</td>
</tr>
<tr>
<td></td>
<td>215,790</td>
<td>595,522</td>
</tr>
<tr>
<td></td>
<td>428,340</td>
<td>1,303,189</td>
</tr>
<tr>
<td></td>
<td>491,225</td>
<td>1,428,101</td>
</tr>
<tr>
<td></td>
<td>389,274</td>
<td>1,115,493</td>
</tr>
<tr>
<td></td>
<td>1,149,770</td>
<td>3,133,399</td>
</tr>
</tbody>
</table>

Table 7.3: Experimental results on FPGA routing problems. C2D_FORGET is our approach for compiling DNNFs without determinism. All timings are in seconds.
compiled for the input CNF. Indeed, the compiled DNNF would not necessarily be
deterministic (as the conjoin operation does not enforce this). However, building
an efficient knowledge compiler based on this approach would require intensive
engineering effort and, to our knowledge, has not been accomplished yet. Our
work, on the other hand, leverages state-of-the-art knowledge compilers as it only
depends on constructing emf formulas. Due to this, one can quickly build an
efficient DNNF compiler, as we have done in this work. Moreover, DNNF could
be exponentially more succinct than its structured subset [PD08], which could
make the mentioned work more restrictive than our presented approach.

Another related work to ours is that of [LM14, LM17], in which the authors
studied the effects of preprocessing CNFs for model counting. They considered
various techniques from the literature and also introduced a few new ones, which
resulted in an efficient preprocessor. Their focus was on constructing CNFs that
are either equivalent or preserving the model count. Our work is based on pre-
processing techniques that generate emf formulas, and targets compiling DNNFs,
as opposed to counting the models.

Finally, [ACM15] studied the problem of projected model counting, in which the
goal is to compute the model count of a formula after forgetting certain variables.
In their setting, auxiliary variables are named as “non-priority” variables. The
main distinction here is that we are not interested in the model counting after
forgetting variables. Because of this, interestingly enough, the forgetting operation
helps in our setting to obtain more compact representations.
7.A Proof of Theorem 24

In this section, we will present the proof of Theorem 24. For that, we first define the primal treewidth of a CNF, based on the notion of a jointree (e.g., [Dar09]).

A jointree for a CNF $\Delta$ is a tree whose vertices are labeled with a subset of variables in $\Delta$ such that the following two conditions hold:

- For each clause $\gamma$ of $\Delta$, there is a vertex whose label contains the variables of $\gamma$;
- If a variable $X$ appears in the labels of two vertices $V_1$ and $V_2$, then each vertex on the path connecting $V_1$ and $V_2$ includes variable $X$ in its label.

The label of a vertex in a jointree is called its cluster. The width of a jointree is the size of its largest cluster minus 1. The primal treewidth of a CNF is the smallest width attained by any of its jointrees, which also corresponds to the treewidth of the primal graph of the CNF. For instance, Figure 7.2 depicts a jointree for $\Delta^b_n$ whose width is 2. We are now ready to prove Theorem 24.

**Proof.** Assume that we applied the BVA transformation on CNF $\Delta$ once, and constructed the CNF $\Delta^1$. So, an auxiliary variable $X$ is added to CNF $\Delta^1$. Consider now the best jointree of $\Delta$ (i.e., the one whose width is $w$). If we add variable $X$ to each label set of its vertices, the resulting tree will clearly be a jointree for $\Delta^1$, with width $w + 1$. So, the treewidth of $\Delta^1$ will be at most $w + 1$. Using the same argument after each application of the BVA transformation, the treewidth will be at most $w + k$ after the $k^{th}$ step. ■
7.B Proof of Theorem 25

In this section, we will present the proof of Theorem 25.

**Proof.** We first show that the treewidth of $\Delta^a_n$ is unbounded (at least $2n$). In the primal graph of $\Delta^a_n$, each vertex will have a degree of $2n$. According to a known result (e.g., [Dar09]), the treewidth of $\Delta^a_n$ is no less than $2n$.

We will now show that the treewidth of $\Delta^b_n$, which can be obtained from $\Delta^a_n$ by the BVA transformation, is bounded (at most 2). Figure 7.2 depicts a jointree for $\Delta^b_n$ whose width is 2. Hence, the treewidth of $\Delta^b_n$ is at most 2.

![Figure 7.2: A jointree for CNF $\Delta^b_n$.](image)
CHAPTER 8

Solving $\text{PP}^\text{PP}$-Complete Problems Using Knowledge Compilation

As discussed in Chapter 2, knowledge compilation has been successfully used to solve Beyond-$\text{NP}$ problems, including some $\text{PP}$-complete and $\text{NP}^\text{PP}$-complete problems for Bayesian networks. In this chapter, we show how knowledge compilation can be used to solve problems in the more intractable complexity class $\text{PP}^\text{PP}$, hence extending its reach in the reduce-then-solve paradigm. In particular, we introduce a special class of SDDs, called constrained SDDs, and develop an algorithm that solves the prototypical $\text{PP}^\text{PP}$-complete problem $\text{MAJMAJSAT}$ in linear time once the problem instance is compiled into this new representation. We propose two techniques to compile constrained SDDs, one of which is based on a new type of vtree operation, called swapping variables, which is of independent interest. We also modify our algorithm to solve a $\text{PP}^\text{PP}$-complete problem that is of practical interest, the same-decision probability (SDP) introduced for Bayesian networks. We empirically compare our proposed approach with the state-of-the-art algorithm for computing the SDP, and show favorable results. The material in this chapter is based on work published in [OCD16].

8.1 Introduction

The complexity class $\text{PP}^\text{PP}$ is highly intractable being at the second level of the counting hierarchy [Wag86]. Despite this difficulty, the $\text{PP}^\text{PP}$ class includes
some interesting and practical AI problems, such as non-myopic value of information [KG09]. Developing effective methods for problems in this class is therefore both significant and difficult.

Our proposed approach for tackling problems in the $\text{PP}^{\text{PP}}$ class will be based on knowledge compilation. The key notion here is to compile problem instances into tractable representations, allowing one to solve such problems efficiently if the compilation is successful. Although knowledge compilation was originally motivated by the need to push much of the computational overhead into an offline compilation phase, it has been increasingly used as a general methodology for computation. In particular, this approach has been successfully used to solve Beyond-$\text{NP}$ problems, including some problems that are complete for the $\text{PP}$ class [Dar01a, Dar03], and the $\text{NP}^{\text{PP}}$ class [HCD06, PD09].

In this chapter, we extend the reach of knowledge compilation techniques to problems in the highly intractable complexity class $\text{PP}^{\text{PP}}$, which contains $\text{NP}^{\text{PP}}$ and can be thought of as its counting analogue. In particular, we introduce a new algorithm for the prototypical $\text{PP}^{\text{PP}}$-complete problem known as $\text{MajMajSat}$. This decision problem is posed with respect to a CNF, asking whether there exists a majority of truth assignments to some variables, under which there is a majority of satisfying truth assignments to the remaining variables [Wag86]. Our algorithm is based on compiling the problem instance into a special class of SDDs. This new class of SDDs we identify in this chapter constrains their structure, allowing one to solve $\text{MajMajSat}$ in time linear in the SDD size.

To show the effectiveness and applicability of our approach, we modify our algorithm to solve a $\text{PP}^{\text{PP}}$-complete problem that is of practical interest: The $\text{Same-Decision Probability}$ (SDP) introduced recently for Bayesian networks [CXD12]. The SDP problem is a value-of-information query that quantifies the robustness of threshold-based decisions. It has been successfully applied as a selection or stopping criterion when making decisions under uncertainty [CCD14, CCD15b,
CCD15a] and in medical diagnosis [GWB14]. Further, it comes with a correspond-
ing exact algorithm [CCD13]. We empirically compare our proposed approach with the state-of-the-art algorithm for computing the SDP, showing favorable re-
sults.

This chapter is organized as follows. We start by providing some technical background (Section 8.2). We then introduce the special class of SDDs and the new MAJMAJSAT algorithm that operates on these SDDs (Section 8.3). This is followed by introducing the SDP problem and a corresponding algorithm based on knowledge compilation (Section 8.4). We then discuss two techniques to compile constraint SDDs (Section 8.5 and Section 8.6). After that, we present empirical results (Section 8.7) and discuss related work (Section 8.8). The appendix contains the proofs of technical results.

8.2 Technical Background

In this chapter, we liberally treat an instantiation of a variable set as the conjunc-
tion of its corresponding literals. Moreover, given instantiations \( x \) and \( y \), we say \( x \) is compatible with \( y \), denoted \( x \sim y \), iff \( x \land y \) is satisfiable.

In Chapter 2, we defined SDDs as respecting vtree nodes. In this chapter, we will employ a slightly different definition of SDDs for the clarity of representation, which is based on normalization:

**Definition 29.** \( \alpha \) is an SDD that is normalized for vtree \( v \) iff:

- \( \alpha = \bot \) or \( \alpha = \top \), where \( v \) is a leaf node;
- \( \alpha = X \) or \( \alpha = \neg X \), where \( v \) is a leaf node labeled by \( X \);
- \( \alpha = \{ (p_1, s_1), \ldots, (p_n, s_n) \} \), where \( v \) is an internal node, \( p_1, \ldots, p_n \) are SDDs that are normalized for \( v^l \), and \( s_1, \ldots, s_n \) are SDDs that are normalized for \( v^r \).
Accordingly, normalization requires primes and subs of a node to respect the left and right children of the current vtree node, respectively, rather than some vtree node in the left and right subtrees of the current vtree node. In the rest of the chapter, we will assume compressed and normalized SDDs. Indeed, this definition is equivalent to the original definition up to some syntactic manipulation preserving polynomial size [Dar11]. Figure 8.1(a) depicts a compressed and normalized SDD for vtree in Figure 8.1(b).

We finally note that if an SDD is normalized for vtree \( v \), then its model count is defined over \( Vars(v) \). For example, the terminal SDDs \( \bot \), \( \top \), \( X \) and \( \neg X \) have model counts of 0, 2, 1, and 1, respectively.

### 8.3 Solving MajMajSat using SDDs

In this section, we will present a new algorithm that solves the prototypical \( \mathsf{PP}^{\mathsf{PP}} \)-complete problem \( \text{MajMajSat} \), using a special type of SDDs. We start with formally defining the problem.
8.3.1 Problem Definition

In the sequel, we will focus on a functional variant of MajMajSat, denoted MMS($f, X, T$), which we introduce as follows:\footnote{Solving the functional version would immediately imply a solution to the decision problem.}

Consider a Boolean function $f(XY)$. Given a threshold $T$, how many instantiations $x$ are there, for which the number of instantiations $y$ that satisfy $f|_x$ is greater than or equal to $T$?

Formally, $MMS(f, X, T)$ can be defined as follows:

$$MMS(f, X, T) = \sum_x [MC(f|_x) \geq T], \quad (8.1)$$

where $[P]$ is an indicator function equal to 1 if $P$ is true and equal to 0 otherwise.

We will next show how to solve MajMajSat in linear-time when the underlying Boolean function is represented as a special type of SDD.

8.3.2 Constrained SDDs

The new class of SDDs will basically have a constrained structure, due to the following special type of vtrees.

**Definition 30.** A vtree node $v$ is $X$-constrained, denoted $v_X$, iff $v$ appears on the right-most path of the vtree and $X$ is the set of variables outside $v$. A vtree is $X$-constrained iff it has an $X$-constrained node.

Figure 8.2 shows an $X$-constrained vtree for $X = \{A, B, D\}$, where vtree node $v = 4$ is the $X$-constrained node. A vtree can have at most one $X$-constrained node. Moreover, an $X$-constrained vtree constrains the variable order obtained by a left-right traversal of the vtree, pushing variables $X$ in front. The reverse is not
necessarily true. For example, the left-right traversal of the vtree in Figure 8.2 puts variables $X = \{A, B\}$ in front, yet the vtree is not $X$-constrained.

To compute $\text{MMS}(f, X, T)$, we will represent the Boolean function $f$ using an $X$-constrained SDD.

**Definition 31.** An SDD is $X$-constrained iff it is normalized for an $X$-constrained vtree. An SDD node is $X$-constrained iff it is normalized for the $X$-constrained vtree node.

Intuitively, an $X$-constrained SDD node corresponds to the conditioning of the SDD $f$ on some instantiation $x$, and will be used to compute the indicator function $[\text{MC}(f|x) \geq T]$.

### 8.3.3 A New Algorithm to Solve MajMajSat

We present the pseudocode of our approach in Algorithm 9, which takes as input a set of variables $X$, a threshold value $T$, and an $X$-constrained SDD $S$, to compute $\text{MMS}(S, X, T)$. The algorithm performs a single bottom-up pass over $S$ (Lines 1–5). For each visited SDD node $\alpha$, the algorithm applies one of several actions, depending on the vtree node $v$ to which $\alpha$ is normalized for.

If vtree node $v$ is not an ancestor of the $X$-constrained vtree node $v_X$, the

---

**Figure 8.2:** An $X$-constrained vtree, where $X = \{A, B, D\}$. 

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{figure82.png}
\caption{An $X$-constrained vtree, where $X = \{A, B, D\}$.}
\end{figure}
Algorithm 9: \(MMS(S, X, T)\)

**Input:**
- \(S\): \(X\)-constrained SDD
- \(X\): set of variables
- \(T\): threshold

**Data:**
- \(vr()\): value registers (one for each SDD node)

**Output:** computes \(MMS(S, X, T)\)

1. foreach SDD node \(\alpha\) in \(S\) (children before parents) do
   2. if \(\alpha\) is a terminal then \(vr(\alpha) \leftarrow MC(\alpha)\)
   3. else \(vr(\alpha) \leftarrow \sum_{(p_i, s_i) \in \alpha} vr(p_i) \times vr(s_i)\)
   4. if \(\alpha\) is \(X\)-constrained then
      5. \(vr(\alpha) \leftarrow 1\) if \(vr(\alpha) \geq T\); else 0

6. return \(vr(S)\)

Algorithm computes the model count of \(\alpha\).

**Lemma 29.** Let \(\alpha\) be an SDD node normalized for \(v\), where \(v\) is not an ancestor of \(v_X\). Line 2 or Line 3 computes \(MC(\alpha)\).

If \(v = v_X\), \(\alpha\) must be equal to \(S|\mathbf{x}\) for some instantiation \(\mathbf{x}\). So, we also compute the indicator function \([MC(\alpha) \geq T]\) (Lines 4–5), and pass either 0 or 1 to the ancestors of \(\alpha\). Because of this, if \(v\) is an ancestor of \(v_X\), the algorithm basically counts the instantiations \(\mathbf{y}\) for which the model count of \(\alpha|\mathbf{y}\) is above the threshold \(T\), where \(\mathbf{Y}\) is the subset of \(X\) appearing in \(v\).

**Lemma 30.** Let \(\alpha\) be an SDD node normalized for \(v\), where \(v\) is an ancestor of \(v_X\) or \(v = v_X\). Then,
\[
vr(\alpha) = \sum_{\mathbf{y}} [MC(\alpha|\mathbf{y}) \geq T],
\]
where \(\mathbf{Y} = Vars(v) \cap X\).
Figure 8.3: Value registers computed by Algorithm 9.

The above cases ensure that the algorithm computes Equation (8.1) at the root of SDD $S$ (Line 6).

**Proposition 10.** Algorithm 9 computes $\text{MMS}(S, X, T)$.

As the algorithm performs a single pass over $S$ (Lines 1–5), and at each node it takes a constant amount of time, the time complexity of Algorithm 9 is linear in the size of $S$.

**Proposition 11.** Algorithm 9 takes time linear in the size of $S$.

We now demonstrate how Algorithm 9 works on constrained SDDs. Consider the SDD $S$ in Figure 8.1(a). This SDD is normalized for the vtree in Figure 8.1(b). This vtree is $X$-constrained for $X = \{A, B\}$, where the root vtree node is $X$-constrained. Given a threshold $T = 3$, Figure 8.3 shows the value registers of SDD nodes computed by Algorithm 9, upon the call $MMS(S, X, T)$, by labeling the nodes with the corresponding values. Accordingly, the root of SDD $S$ returns 1, meaning that there exists only one instantiation $x$ of variables $X$ such that the model count of $S|x$ exceeds the given threshold $T = 3$.

We close this section with the following remark. When the Boolean function is represented as a general SDD (serving the role of a PP oracle), $\text{MajMajSat}$ would be PP-hard. This is still true when we use other well-known compilation languages that support polynomial-time model counting, such as d-DNNFs [Dar01a]
and OBDDs [Bry86]. To solve the problem in linear-time, however, we need an additional property, as in X-constrained vtrees. We are not aware if there is a weaker property that would suffice.

8.4 Computing the SDP using SDDs

We will now modify our algorithm to solve another PP\textsuperscript{pp}-complete problem, which is of practical interest: The Same-Decision Probability (SDP) for Bayesian networks [CXD12]. The input to this problem is a probability distribution represented by the Bayesian network, together with some variable sets and a threshold. Intuitively, the SDP is used to quantify the robustness of decisions against new evidence. That is, given initial evidence $e$, one makes a decision $d$ based on whether $\Pr(d|e)$ surpasses a given threshold $T$. The SDP is then the probability that this decision would stay the same after observing the state of new evidence (which is also given as input). We will actually define an abstraction of this problem in which the distribution is represented by a weighted Boolean function. This abstraction will facilitate the computation of SDP through compilation into SDDs.

8.4.1 SDP on Weighted Boolean Functions

We start by defining weighted Boolean functions, which simply augment a Boolean function with a weight function.

**Definition 32.** A weighted Boolean function is a pair $(f, W)$ where $f$ is a Boolean function over variables $Z$ and $W$ is a weight function that maps literals of $Z$ to real numbers.

Given a weighted Boolean function $(f, W)$, one is typically interested in computing
its weighted model count, which is formally defined as follows:

$$\sum_{\mathbf{z} \models f} \left( \prod_{\ell \in \mathbf{z}} W(\ell) \right).$$

That is, the weighted model count of $f$ is the summation of the weights of the models of $f$, where the weight of a model is the product of its literals’ weights. The weighted model count subsumes the model count when the weight function $W$ assigns the weight 1 to each literal.

We also define the weighted model count with respect to an instantiation $\mathbf{x}$, also called evidence $\mathbf{x}$:

$$\phi_{(f,W)}(\mathbf{x}) = \sum_{\mathbf{z} \models f, \mathbf{z} \sim \mathbf{x}} \left( \prod_{\ell \in \mathbf{z}} W(\ell) \right).$$

That is, the weighted model count of $f$ under evidence $\mathbf{x}$ is the summation of the weights of the models of $f$ compatible with $\mathbf{x}$. We will omit $W$ from the subscript whenever it is clear from the context, and write $\phi_f(\mathbf{x})$ instead. Under no evidence (i.e., $\mathbf{x} \equiv \top$), we often drop $\mathbf{x}$ and write $\phi_f$. In this case, $\phi_f$ reduces to the weighted model count.

We finally define the conditional weighted model count:

$$\phi_f(\mathbf{x} \mid \mathbf{y}) = \frac{\phi_f(\mathbf{x} \mathbf{y})}{\phi_f(\mathbf{y})}.$$ 

Note here that $\phi_f(\cdot \mid \top)$ is a probability distribution. Moreover, $\phi_f(\cdot \mid \mathbf{y})$ is a probability distribution conditioned on instantiation $\mathbf{y}$. This shows how a weighted Boolean function can be used to represent a probability distribution, allowing us to define the same-decision probability on weighted Boolean functions.

**Definition 33.** Consider a weighted Boolean function $(f, W)$. Let $\mathbf{E}, \mathbf{H}$ and $\{D\}$ be mutually disjoint variables of $f$. Given an instantiation $d$, a threshold $T$, and an instantiation $\mathbf{e}$, the same-decision probability is defined as follows:

$$SDP_f(d, \mathbf{H}, \mathbf{e}, T) = \sum_{\mathbf{h}} [\phi_f(d \mid h \mathbf{e}) \geq T] \phi_f(h \mid \mathbf{e}).$$ (8.2)
We remark that the classical definition of SDP on Bayesian networks is based on the probability distribution defined by the Bayesian network [CXD12].\textsuperscript{2} Equation (8.2), however, replaces that distribution by its weighted Boolean function representation. Similarly to its Bayesian network analogue, the SDP on weighted Boolean functions is highly intractable, assuming a CNF representation of the function.

**Theorem 26.** The problem of deciding whether the SDP on a weighted CNF is greater than a number $p$ is \textbf{PP$^\text{PP}$}-complete.

As we shall see next, if a Boolean function is represented by an $H$-constrained SDD, the SDP problem becomes tractable. Indeed, as long as the set of variables $H$ does not change, each instance of SDP with different parameters $d, e, T$ can be solved in linear-time in the SDD size. That is, our knowledge compilation approach would effectively solve exponentially many queries in linear-time.

### 8.4.2 A New Algorithm to Compute SDPs

We now present our method to compute the SDP on weighted Boolean functions, using constrained SDDs. The pseudocode of our approach is described in Algorithm 10, which takes as input an SDP instance with parameters $d, H, e, T$ and an $H$-constrained SDD $S$, to compute $\text{SDP}_S(d, H, e, T)$. The algorithm is a slight modification of Algorithm 9 presented earlier. It performs two bottom-up passes over $S$ and maintains two value registers per SDD node. This is because it needs to compute (conditional) weighted model counts under evidence.

Given an SDD node $\alpha$ normalized for vtree node $v$ and evidence $e$, we let $e_v$ denote the subset of the instantiation $e$ that pertains to the variables of vtree $v$.

\textsuperscript{2}For Bayesian networks, the SDP is used to quantify the robustness of decisions against new evidence. That is, given initial evidence $e$, one makes a decision based on whether $\Pr(d|e) \geq T$. The SDP is then the probability that this decision would stay the same after observing the state of variables $H$ (new evidence).
Algorithm 10: \( SDP(d, H, e, T, S) \)

**Input:**
- \( d \) : hypothesis
- \( H \) : query variables
- \( e \) : evidence
- \( T \) : threshold
- \( S \) : (root) \( H \)-constrained SDD

**Data:**
- \( vr_1(), vr_2() \) : value registers (one for each SDD node)

**Output:** computes \( SDP_S(d, H, e, T) \)

```plaintext
1 foreach SDD node \( \alpha \) in \( S \) (children before parents) do
2     if \( \alpha \) is a terminal then
3         \( vr_1(\alpha) \leftarrow \phi_\alpha \) if \( \alpha \sim e \); else 0
4     else \( vr_1(\alpha) \leftarrow \sum_{(p_i, s_i) \in \alpha} vr_1(p_i) \times vr_1(s_i) \)
5 foreach SDD node \( \alpha \) in \( S \) (children before parents) do
6     if \( \alpha \) is a terminal then
7         \( vr_2(\alpha) \leftarrow \phi_\alpha \) if \( \alpha \sim d e \); else 0
8     else \( vr_2(\alpha) \leftarrow \sum_{(p_i, s_i) \in \alpha} vr_2(p_i) \times vr_2(s_i) \)
9     if \( \alpha \) is \( H \)-constrained then
10        \( vr_2(\alpha) \leftarrow vr_1(\alpha) \) if \( \frac{vr_2(\alpha)}{vr_1(\alpha)} \geq T \); else 0
11 \( \phi(e) \leftarrow vr_1(S) \)
12 \( Q \leftarrow vr_2(S) \)
13 return \( \frac{Q}{\phi(e)} \)
```
Algorithm 10 then computes $\phi_\alpha(e_v)$ for each node $\alpha$ in $S$, during the first pass (Lines 1–4). The result is cached in the register $vr_1(\alpha)$.

**Lemma 31.** Let $\alpha$ be an SDD node normalized for vtree $v$. Then, $vr_1(\alpha) = \phi_\alpha(e_v)$.

In the second pass (Lines 5–10), the algorithm mimics Algorithm 9. First, if $v$ is not an ancestor of $v_H$, it simply computes a weighted model count.

**Lemma 32.** Let $\alpha$ be an SDD node normalized for vtree $v$, where $v$ is not an ancestor of $v_H$. Lines 7–8 compute $\phi_\alpha(d e_v)$ if $D$ is contained in vtree $v$, and $\phi_\alpha(e_v)$ otherwise.

Next, if $v = v_H$, then $\alpha$ must be equal to $S|h$ for some instantiation $h$. Here, we also compute the indicator function $[\phi_\alpha(d|e) \geq T]$ (Lines 9–10) and pass either 0 or $\phi_\alpha(e)$ to the ancestors of $\alpha$. This way, if $v$ is an ancestor of $v_H$, it basically computes the following quantity:

$$\sum_y \phi_\alpha(d|y, e) \geq T \phi_\alpha(y e),$$

where $Y$ is the subset of $H$ appearing in $v$.

**Lemma 33.** Let $\alpha$ be an SDD node normalized for vtree $v$, where $v$ is an ancestor of $v_H$ or $v = v_H$. Then,

$$vr_2(\alpha) = \sum_y [\phi_\alpha(d|y e) \geq T] \phi_\alpha(y e),$$

where $Y = Vars(v) \cap H$.

We now obtain the following quantity at the root of SDD $S$:

$$Q = \sum_h [\phi_S(d|h e) \geq T] \phi_S(h e).$$

The quantity $Q$ is not equal to the SDP (note $\phi_S(h, e)$ instead of $\phi_S(h | e)$). Dividing $Q$ by $\phi_S(e)$, which is computed by the first pass, gives the desired result (Line 13).
Proposition 12. Algorithm 10 computes $SDP_S(d, H, e, T)$.

The algorithm takes two passes over SDD $S$ (Lines 1–4 and Lines 5–10). During each pass, the work it performs at each SDD node takes a constant amount of time. Hence, the time complexity of Algorithm 10 is linear in the size of $S$.

Proposition 13. Algorithm 10 takes time linear in the size of $S$.

8.5 Compiling X-Constrained SDDs

Our algorithms require the representation of Boolean functions as constrained SDDs, which also require the construction of constrained vtrees. Since the SDD size depends critically on the corresponding vtree, identifying good constrained vtrees is quite critical for compiling successfully. Moreover, once compilation is completed, further queries can be answered in time linear in the SDD size. Hence, the smaller the compiled SDDs are, the more efficient further inference will be. We will next describe two methods for obtaining X-constrained vtrees that tend to yield smaller SDD sizes.

8.5.1 Method 1 – Constrained Search

Typically, there are two main approaches for generating vtrees. The first approach, which is static, identifies an appropriate vtree before compilation starts. This method requires a preprocessing step of the Boolean function representation. The second approach, which is dynamic, searches for an appropriate vtree during the compilation process. That is, while compiling the SDD, one can use a search algorithm that tries to identify vtrees leading to smaller SDD sizes. Although this search might be costly, it is needed most of the time to successfully finish compilation.

In our first method to obtain X-constrained vtrees, we will combine static
and dynamic approaches as follows. Our initial vtree will be right-linear, where variables \( \mathbf{X} \) appear first in the left-right traversal of the vtree.\(^3\) Here, one can use some heuristic method to find a variable ordering for variables \( \mathbf{X} \) and another for the remaining variables.\(^4\) By combining these two variable orderings, one can obtain an \( \mathbf{X} \)-constrained vtree, which is right-linear. Then, during compilation, we can employ a dynamic vtree search algorithm as long as the search algorithm ensures that the new vtree remains \( \mathbf{X} \)-constrained. In our case, we will modify the search algorithm introduced by [CD13b]. This algorithm essentially navigates the search space of vtrees by applying three different vtree operations on vtree nodes, namely, right rotation, left rotation, and children swapping. These operations are depicted in Figure 8.4. To make sure the new vtree will remain \( \mathbf{X} \)-constrained, all we need is to restrict the use of some vtree operations at certain vtree nodes. In particular, the restrictions below suffice for our purpose:

- Right rotation can be done on any vtree node.

- Left rotation can be done on any vtree node unless it is the \( \mathbf{X} \)-constrained vtree node.

- Children swapping can be done on any vtree node unless it is an ancestor of the \( \mathbf{X} \)-constrained vtree node.

\(^3\)In this case, the SDD would correspond to an OBDD.

\(^4\)We use the minfill heuristic which is commonly used in the Bayesian network literature [Dar09].
We note that the restrictions listed above would apply to any vtree search algorithm that is based on the vtree operations mentioned. Hence, any advances on such algorithms might potentially improve this method.

8.5.2 Method 2 – Moving Variables by Swapping

The approach discussed above basically restricts the whole compilation process to the space of \( X \)-constrained vtrees, which would be limited compared to the space of all possible vtrees. Thus, this restriction could make compilation harder in some cases, which otherwise can be compiled easily when no restriction is imposed on vtrees. This observation is the basis of our second approach. Here, instead of restricting the whole compilation process to constrained vtrees, we first relax the problem and attempt to compile a general SDD, which is not necessarily constrained. This would definitely be much more efficient than compiling a constrained SDD directly. Once we obtain a general SDD, we then try to convert it into an \( X \)-constrained SDD. This is achieved by moving variables \( X \) in the vtree (and thus obtaining an \( X \)-constrained vtree), as illustrated next.

Suppose that we compiled a general SDD, which is normalized for the vtree \( v \) in Figure 8.5(a). Assume further that we want to obtain an \( X \)-constrained vtree, for \( X = \{C, D\} \). Clearly, vtree \( v \) is not \( X \)-constrained. To make it \( X \)-constrained, we suggest the following method. We first move variable \( D \) to the top of this vtree, and obtain the vtree \( v_D \) in Figure 8.5(b). Vtree \( v_D \) is also not \( X \)-constrained, but it is a \( \{D\} \)-constrained vtree which is closer. We then move variable \( C \) to the top of vtree \( v_D \), obtaining the vtree \( v_{CD} \) in Figure 8.5(c). Now, vtree \( v_{CD} \) is \( X \)-constrained. Thus, one can always convert an arbitrary vtree into an \( X \)-constrained vtree by moving variables in \( X \) consecutively, as we just explained. Indeed, moving variables in the vtree would change the partitions underlying the corresponding SDD. Therefore, we also need a method that moves variables while adjusting the SDD accordingly, which is discussed next.
Moving variables: The vtree operations shown in Figure 8.4 can be used to navigate the space of all vtrees in a systematic way [CD13b]. Thus, theoretically, one can also use them to move variables in a vtree. However, this will be impractical as one may need to perform too many of those vtree operations. Further, constructing an algorithm that moves a variable to some position using only those three operations seems nontrivial. Therefore, we propose here a new vtree primitive that allows one to easily move variables in a vtree: swapping two arbitrary variables in a vtree.\(^5\) Before explaining how to adjust SDD nodes with respect to the new vtree primitive, we next explain how to employ it to move a variable to a new position in a vtree.

Basically, we can move a variable \(X\) to an arbitrary position in a vtree using the following steps: (1) add a new variable \(X'\) to the position \(X\) is going to be moved, (2) use the new primitive to swap \(X\) and \(X'\) in the vtree, and (3) remove \(X'\) from the vtree.

As mentioned earlier, performing these steps would require adjusting SDD nodes, since the underlying partitions would change. Here, steps (1) and (3) are trivial as adding or removing an unused variable will not change the structure of SDD nodes, except trivially. Step (2), however, requires the swapping primitive, which can be obtained as follows.

\(^5\)Note the difference from swapping children which only swaps the children of a vtree node.
Swapping variables: Suppose that we want to swap variables $X$ and $Y$ in a vtree. This only requires adjusting SDD nodes that depend on either $X$ or $Y$ (or both). Note that those SDD nodes can only be normalized for vtree nodes in one of the following categories:

- Lowest common ancestor (lca) $v$ of $X$ and $Y$;
- Vtree nodes on the path from $v'$ to $X$ (denoted $v' \mapsto X$);
- Vtree nodes on the path from $v''$ to $Y$ (denoted $v'' \mapsto Y$).

Thus, we can adjust those SDD nodes by converting between equivalent partitions in each category, which we explain next. In the following, assume that $\text{vars}(v') = A \cup \{X\}$ and $\text{vars}(v'') = B \cup \{Y\}$ ($A$ and $B$ being possibly empty).

**SDDs normalized for the lca $v$:** Let $\alpha$ be an SDD node normalized for the lca $v$, equal to $\{(p_1, s_1), \ldots, (p_n, s_n)\}$. To adjust $\alpha$ with respect to swapping variables $X$ and $Y$, we need to consider its dependence on $X$ and $Y$.

Assume that $\alpha$ depends on both $X$ and $Y$. In this case, we need a conversion from an $(AX, BY)$-partition to an equivalent $(AY, BX)$-partition. This can be done by utilizing the following extended version of the Shannon decomposition:

$$\alpha \equiv (X \land Y \land \alpha | XY) \lor (X \land \neg Y \land \alpha | X \neg Y) \lor (\neg X \land Y \land \alpha | \neg XY) \lor (\neg X \land \neg Y \land \alpha | \neg X \neg Y).$$

Note that each term above can be written as the following equivalent $(AY, BX)$-
partition, which will be denoted by \( P_1, P_2, P_3, \) and \( P_4 \), respectively:

\[
P_1 = \{(p_i|X \land Y, s_i|Y \land X) \mid p_i|X \neq \bot, i \in [1, n]\} \cup \{(\neg Y, \bot)\},
\]

\[
P_2 = \{(p_i|X \land \neg Y, s_i|\neg Y \land X) \mid p_i|X \neq \bot, i \in [1, n]\} \cup \{(Y, \bot)\},
\]

\[
P_3 = \{(p_i|\neg X \land Y, s_i|Y \land \neg X) \mid p_i|\neg X \neq \bot, i \in [1, n]\} \cup \{(\neg Y, \bot)\},
\]

\[
P_4 = \{(p_i|\neg X \land \neg Y, s_i|\neg Y \land \neg X) \mid p_i|\neg X \neq \bot, i \in [1, n]\} \cup \{(Y, \bot)\}.
\]

Therefore, the following Cartesian product of the partitions above would yield an equivalent \((A_Y, B_X)\)-partition of \( \alpha \), which may not be compressed:

\[
\alpha \equiv P_1 \times P_2 \times P_3 \times P_4. \tag{8.3}
\]

Note that if \( \alpha \) does not depend on both \( X \) and \( Y \), we can still use Equation (8.3) to compute the corresponding equivalent partition. Yet, it can be simplified more, as shown below.

Assume that \( \alpha \) depends on \( X \) but not on \( Y \). So, we need a conversion from an \((A_X, B)\)-partition to an equivalent \((A, B_X)\)-partition. In this case, the extended Shannon decomposition would simplify to the Shannon decomposition:

\[
\alpha \equiv (X \land \alpha|X) \lor (\neg X \land \alpha|\neg X).
\]

Each term can now be written as the following equivalent \((A, B_X)\)-partition (in the order of appearance):

\[
P_5 = \{(p_i|X, s_i \land X) \mid p_i|X \neq \bot, i \in [1, n]\},
\]

\[
P_6 = \{(p_i|\neg X, s_i \land \neg X) \mid p_i|\neg X \neq \bot, i \in [1, n]\}.
\]

Hence, the following Cartesian product would be an equivalent \((A, B_X)\)-partition of \( \alpha \), which would be compressed if \( P_5 \) and \( P_6 \) are both compressed:

\[
\alpha \equiv P_5 \times P_6. \tag{8.4}
\]
Assume that $\alpha$ depends on $Y$ but not on $X$. So, we need a conversion from an $(A, BY)$-partition to an equivalent $(AY, B)$-partition. As in the previous case, the extended Shannon decomposition would simplify to the Shannon decomposition:

$$\alpha \equiv (Y \land \alpha|Y) \lor (\neg Y \land \alpha|\neg Y).$$

Each term can now be written as the following equivalent $(AY, B)$-partition (in the order of appearance):

$$P_7 = \{(p_i \land Y, s_i|Y) \mid i \in [1, n]\} \cup \{(\neg Y, \bot)\},$$
$$P_8 = \{(p_j \land \neg Y, s_j|\neg Y) \mid i \in [1, n]\} \cup \{(Y, \bot)\}.$$

Thus, disjoining the partitions above would yield an equivalent $(AY, B)$-partition of $\alpha$, which may not be compressed:

$$\alpha \equiv P_7 \lor P_8. \quad (8.5)$$

If $\alpha$ does not depend on neither $X$ nor $Y$, then $\alpha$ keeps having the same structure since its underlying partition does not change after the swapping (an $(A, B)$-partition).

**SDDs normalized for vtrees on the path $v^t \mapsto X$:** Let $\alpha$ be an SDD node normalized for a vtree on the path $v^t \mapsto X$. Note that $\alpha$ cannot depend on $Y$, but may depend on $X$. If it does not depend on $X$, then no adjustment is needed. Otherwise, $\alpha$ can be written as $\{(\alpha, \top), (\neg \alpha, \bot)\}$, which is the compressed $(AX, B)$-partition. Thus, we can use Equation (8.4), which would simplify to the following compressed partition (after removing false primes):

$$\{(\alpha|X \land \alpha|\neg X, \top), (\alpha|X \land \neg \alpha|\neg X, X),$$
$$\neg \alpha|X \land \alpha|\neg X, \neg X), (\neg \alpha|X \land \neg \alpha|\neg X, \bot)\}. \quad (8.6)$$

**SDDs normalized for vtrees on the path $v^r \mapsto Y$:** Let $\alpha$ be an SDD node normalized for a vtree on the path $v^r \mapsto Y$. Note that $\alpha$ cannot depend on $X$, but
may depend on $Y$. If it does not depend on $Y$, then no adjustment is needed. Otherwise, $\alpha$ can be seen as $(\top, \alpha)$, which is the compressed $(\mathbf{A}, \mathbf{B}Y)$-partition. Thus, we can utilize Equation (8.5), which would simplify to the following compressed partition:

$$\{(Y, \alpha|Y), (\neg Y, \alpha|\neg Y)\}. \tag{8.7}$$

We note that variable $X$ will be unused for our purpose (i.e., no SDD node will depend on it). In this case, it will be sufficient to use Equations (8.5) and (8.7). However, our implementation considers all cases, allowing variable $X$ to be used as well. Therefore, this new vtree primitive can also be employed in a different context, such as in building a more effective dynamic vtree search algorithm (note that it is not trivial to move variables using existing vtree primitives).

We now put it all together and present the pseudocode of the variable swapping primitive in Algorithm 11.

We finally discuss incorporating a dynamic vtree search algorithm while moving variables, which would be necessary in practice. Note that moving variables as explained earlier would result in a vtree which is right-linear up to the $X$-constrained vtree node (e.g., see the path from the root to the marked vtree node in Figure 8.5(c)). This would obviously limit the SDDs we can construct. Thus, as we move variables, we will invoke a dynamic vtree search algorithm if the SDD size increases more than a certain percentage (we set this to 75% in the experiments). Indeed, this search algorithm has to ensure that the final vtree is constrained. For this reason, we use the same (constrained) vtree search algorithm as discussed earlier. In particular, suppose that so far we moved variables $X' \subseteq X$ to the top of vtree, making the current vtree $X'$-constrained. If a vtree search algorithm is invoked, then we would make sure that the new vtree remains $X'$-constrained. For example, we can obtain the vtree in Figure 8.5(d) after invoking the search on the vtree in Figure 8.5(c). Overall, we use Algorithm 12 to convert general SDDs into constrained SDDs, which employs the techniques introduced in this section.
Algorithm 11: \textit{swap}(T, X, Y)

\textbf{Input:}

\begin{itemize}
  \item $T$: vtree (each node associated with normalized SDDs)
  \item $X, Y$: variables where $X$ appears before $Y$ in vtree $T$
\end{itemize}

\textbf{Output:} swaps $X$ and $Y$ in $T$ (while adjusting SDDs)

\begin{algorithmic}
\STATE $v \leftarrow \text{lca of } X \text{ and } Y \text{ in vtree } T$
\FOR{\textit{SDD node } $\alpha$ \textit{normalized for a vtree on path } $v \mapsto X$}
  \STATE \textbf{if } $\alpha$ \textit{depends on } $X$ \textbf{then} \textit{save } $\alpha|X$ \textit{and } $\alpha|\neg X$
\ENDFOR
\FOR{\textit{SDD node } $\alpha$ \textit{normalized for a vtree on path } $v' \mapsto Y$}
  \STATE \textbf{if } $\alpha$ \textit{depends on } $Y$ \textbf{then} \textit{save } $\alpha|Y$ \textit{and } $\alpha|\neg Y$
\ENDFOR
\STATE swap variables $X$ and $Y$ in vtree $T$
\FOR{\textit{SDD node } $\alpha$ \textit{normalized for vtree } $v$}
  \STATE \textbf{if } $\alpha$ \textit{depends on } $X$ \textbf{or } $Y$ \textbf{then}
  \STATE \quad adjust $\alpha$ using saved conditionings and one of the equations (8.3), (8.4), and (8.5).
\ENDFOR
\FOR{\textit{SDD node } $\alpha$ \textit{normalized for a vtree on path } $v' \mapsto Y$}
  \STATE // \textit{children before parents}
  \STATE \textbf{if } $\alpha$ \textit{depends on } $X$ \textbf{and is alive} \textbf{then}
  \STATE \quad adjust $\alpha$ using saved conditionings and Equation (8.6).
\ENDFOR
\FOR{\textit{SDD } $\alpha$ \textit{normalized for a vtree on path } $v'' \mapsto X$}
  \STATE // \textit{children before parents}
  \STATE \textbf{if } $\alpha$ \textit{depends on } $Y$ \textbf{and is alive} \textbf{then}
  \STATE \quad adjust $\alpha$ using saved conditionings and Equation (8.7).
\ENDFOR
\end{algorithmic}
Algorithm 12: move($X, S$)

**Input:**
- $X$: set of variables
- $S$: (root) SDD

**Output:** convert $S$ into an $X$-constrained SDD

1. $v \leftarrow$ (root) vtree node $S$ is normalized for
2. **foreach** variable $X$ in $X$ **do**
   3. add a new variable $X'$ to the top of vtree $v$
   4. $swap(v, X', X)$
   5. remove variable $X'$ from vtree $v$
   6. **if** SDD size increases more than 75% **then**
      7. invoke (constrained) vtree search algorithm

8.6 Compiling Bayesian Networks into Weighted SDDs

We defined the SDP problem on weighted Boolean functions, while the original SDP problem is defined on Bayesian networks. Our experiments in the next section assume a Bayesian network input. We therefore need to capture the probability distribution of a Bayesian network as a weighted Boolean function, represented by an SDD. We discuss this process briefly in this section as it is described in detail by [Dar02] and [CKD13]. The reader is also referred to [Dar09] for an introduction to Bayesian networks.

Suppose we are given a Bayesian network $\mathcal{N}$ that induces a probability distribution $\Pr$ over variables $Z$. The weighted Boolean function $(f, W)$ which represents distribution $\Pr(Z)$ has the following variables and weights.

**Indicator variables:** Function $f$ has an *indicator variable* $I_x$ for each value $x$ of variable $X$ in $Z$. Moreover, for each variable $I_x$, the corresponding weights are $W(I_x) = 1$ and $W(\neg I_x) = 1$. 
**Parameter variables:** Function $f$ has a *parameter variable* $P_{x|u}$ for each Bayesian network parameter $\theta_{x|u}$. Moreover, for each variable $P_{x|u}$, the corresponding weights are $W(P_{x|u}) = \theta_{x|u}$ and $W(\neg P_{x|u}) = 1$.

Using the above variables, [Dar02] describes an efficient CNF representation of the Boolean function $f$. Hence, one can compile this CNF into a constrained SDD for the purpose of computing the SDP. Moreover, we can also use a more recent approach which compiles the Boolean function $f$ directly into an SDD, therefore bypassing the construction of a CNF [CKD13]. See also [Rot96] for a theoretical treatment of reducing probabilistic inference to weighted model counting, and [SBK05, CD08] for further practical approaches and reviews of encoding Bayesian networks into weighted CNFs.

### 8.7 Experiments

We now empirically evaluate our proposed SDD-based approach for solving $PP^{\Sigma_2}$-complete problems. To our knowledge, the only existing system for solving $PP^{\Sigma_2}$-complete problems is the SDP algorithm of [CCD13], which is implemented in a beta version of the Bayesian network modeling and reasoning system, SamIam.\(^6\) Hence, we compare our approach to SamIam’s, in the task of computing SDPs in Bayesian networks. Note that both approaches produce exact solutions, hence our evaluation is based on the efficiency of computing the SDP.

We evaluated our systems using the Deterministic QMR (DQMR) benchmarks,\(^7\) which are deterministic versions of the classical diagnostic QMR benchmarks. These are two-layer networks where a top layer represents diseases and the bottom layer represents symptoms; logical-ORs are used (by DQMR) instead of noisy-ORs (by QMR) to represent the disease-symptom interactions. The DQMR

---

\(^6\)SamIam is available at [http://reasoning.cs.ucla.edu/samiam](http://reasoning.cs.ucla.edu/samiam). We obtained a beta version from the authors.

\(^7\)[http://www.cs.rochester.edu/users/faculty/kautz/Cachet](http://www.cs.rochester.edu/users/faculty/kautz/Cachet).
benchmark consists of 120 Bayesian networks. We evaluated each network with 3 settings on the number of query variables \( H \): 10, 20 and 30. For each setting, we created 5 SDP problems at random, where the decision variable and query variables were chosen at random, and 10 evidence variables were selected at random and set to true (which is the more challenging setting for logical-OR networks). To compute SDPs using our algorithm, we compiled Bayesian networks into SDDs as discussed in Section 8.6. The constrained vtrees are constructed as discussed in Section 8.5. The SDP algorithm of SamIam was run with default settings. For each problem, we employed the standard pruning techniques on the corresponding Bayesian network (e.g., see [Dar09] for details). Finally, our experiments were performed on a 2.6GHz Intel Xeon E5-2670 CPU with a 1 hour time limit and a memory limit of 12GB RAM.

Table 8.1 highlights the results. Here, directKC is the method where we obtained constrained SDDs directly, by augmenting the vtree search algorithm (i.e., Method 1 in Section 8.5). The method relaxKC_BU first compiles a general SDD, and then converts it into a constrained SDD by moving variables (i.e., Method 2 in Section 8.5). Both directKC and relaxKC_BU use the recently introduced approach that compiles Bayesian networks directly into SDDs, bypassing the encoding of a CNF [CKD13]. This approach is based on bottom-up compilation. The method relaxKC_TD is similar to relaxKC_BU, except that it uses the top-down SDD compiler introduced in Chapter 6\(^8\), which requires a CNF encoding of the Bayesian network.\(^9\) The method SamIam is the SDP algorithm of [CCD13]. In the table, we report the average number of instances solved for each query setting, together with the (average) total time spent on solving instances.

We now make a few observations. First, although directKC solved more in-
<table>
<thead>
<tr>
<th>Methods</th>
<th>$H: 10$</th>
<th>$H: 20$</th>
<th>$H: 30$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solved instances</td>
<td>Total time (sec)</td>
<td>Solved instances</td>
</tr>
<tr>
<td>directKC</td>
<td>119</td>
<td>4456</td>
<td>98</td>
</tr>
<tr>
<td>relaxKC_BU</td>
<td>120</td>
<td>89</td>
<td>120</td>
</tr>
<tr>
<td>relaxKC_TD</td>
<td>120</td>
<td>16</td>
<td>120</td>
</tr>
<tr>
<td>SAMIAM</td>
<td>70</td>
<td>3517</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 8.1: Experimental results on DQMR-networks.

stances than SAMIAM for $H = 10$ and $H = 20$, this changes for $H = 30$. Next, the methods relaxKC_BU and relaxKC_TD outperformed both directKC and SAMIAM in each query setting, solving more instances using less amount of time. Finally, we remark that the timing for the SDD-based approaches includes both the compilation time, and the SDD evaluation time to compute the SDP. In practice, compilation would be performed in an offline phase. Further, evaluation time on an SDD is relatively negligible. This is quite significant, as one expects to use the SDP in an online fashion in practice (one would continue to make observations, and re-compute the SDP until the SDP is high enough, and further observations are unlikely to be relevant). In contrast, the SDP algorithm of SAMIAM is in general invoked from scratch.

Finally, we want to stress the important role that SDDs and the dynamic vtree search played in our experiments. For that, we disabled the dynamic search and ran the experiment again. In this case, the compiled SDDs would be OBDDs as the initial vtrees were right-linear. However, this approach finished compilation for a very few instances (5 out of 360).\footnote{As the approach in [CKD13] is based on SDDs, using dynamic search while ensuring that the result is an OBDD is nontrivial. For a detailed empirical analysis on SDDs and OBDDs, see [CD13b] who report orders-of-magnitude better performance in favour of SDDs.}
8.8 Related Work

$PP^{PP}$-complete problems, such as MAJMAJSAT and SDP, are particularly challenging as they are still $PP$-hard given access to a $PP$-oracle. Analogously, $NP^{PP}$-complete problems, such as the MAP problem in Bayesian networks, are still $NP$-hard given access to a $PP$-oracle [Par02]. In this context, search-based MAP algorithms have been proposed that assume an oracle for exact inference in Bayesian networks, which is $PP$-complete [PD03]. Knowledge compilation techniques have also been employed here [HCD06, PD09], where Bayesian networks were compiled into d-DNNFs, which were in turn used as oracles. Further, a relaxed notion of constraining a d-DNNF structure was used to produce tighter upper-bounds on the MAP solution, which improves the efficiency of search. This notion of constraining a d-DNNF is similar, but distinct from the notion of a constrained vtree which we introduced in this paper (SDDs and vtrees have stronger semantics). Further, we remark that since $NP^{PP} \subseteq PP^{PP}$, we consider more demanding problems, compared to the above prior work; in fact, our approach can be leveraged to solve MAP as well (we do not pursue this here). Another distinction is that the above works compiled Bayesian networks to d-DNNFs, but through an intermediate CNF representation. In contrast, we can bypass this step using SDDs, which can be more efficient in practice [CKD13]. This is enabled by an efficient $\text{Apply}$ operation, which SDDs support, whereas d-DNNFs do not [DM02].
8.A Soundness of Algorithm 9

We show the soundness of Algorithm 9 whose proof was outlined earlier. We start with a few complementary lemmas.

Lemma 34. Let \( f = \{(p_1, s_1), \ldots, (p_n, s_n)\} \) be an \((X, Y)\)-partition. Then, for each \( x \models p_i(X), s_i(Y) = f(XY)|x \).

**Proof.** Take some instantiation \( x \models p_i(X) \). Since primes are mutually exclusive, \( x \not\models p_j(X) \) for \( j \neq i \). Thus, \( f|_x \) can be obtained as follows:

\[
f|_x = (p_1|_x \land s_1|x) \lor \ldots \lor (p_n|_x \land s_n|x)
\]

\[
= (\bot \land s_1) \lor \ldots \lor (\top \land s_i) \lor \ldots \lor (\bot \land s_n)
\]

\[
= s_i.
\]

\[\square\]

Lemma 35. Let \( f = \{(p_1, s_1), \ldots, (p_n, s_n)\} \) be an \((X, Y)\)-partition and \( e = e' e'' \) be evidence over \( f \) where \( e' \) and \( e'' \) are the (partial) instantiations of \( X \) and \( Y \), respectively. Then, \( \phi_f(e) = \sum_i \phi_{p_i}(e') \times \phi_{s_i}(e'') \).

**Proof.** As \( f \) is an \((X, Y)\)-partition, the sub-functions \( p_i(X) \) and \( s_i(Y) \) are defined over disjoint sets of variables. So, \( \phi_{p_i \land s_i}(e) = \phi_{p_i}(e') \times \phi_{s_i}(e'') \). Since \( p_i \)'s are mutually exclusive, \( (p_i \land s_i) \land (p_j \land s_j) = \bot \) for \( i \neq j \). Hence, \( \phi_f(e) = \sum_i \phi_{p_i}(e') \times \phi_{s_i}(e'') \).

\[\square\]

Corollary 5. Let \( f = \{(p_1, s_1), \ldots, (p_n, s_n)\} \) be an \((X, Y)\)-partition. Then, \( MC(f) = \sum_i MC(p_i) \times MC(s_i) \).

**Proof.** Follows from Lemma 35, as \( MC(f) = \phi_f(\top) \) when the weight function \( W \) assigns 1 to each literal.

\[\square\]

We are now ready to prove Lemma 29 and Lemma 30, which perform case analysis on vtrees.
Lemma 29. Let $\alpha$ be an SDD node normalized for $v$, where $v$ is not an ancestor of $v_X$. Line 2 or Line 3 computes $MC(\alpha)$.

Proof. If $\alpha$ is a terminal SDD, Line 2 clearly computes $MC(\alpha)$. Suppose that $\alpha$ is a decomposition SDD. Let $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$. As $v$ is not an ancestor of $v_X$, Line 5 will never be executed for any descendants of $\alpha$. So, Line 3 computes $\sum_i MC(p_i) \times MC(s_i)$, which is equal to $MC(\alpha)$ by Corollary 5. 

Lemma 30. Let $\alpha$ be an SDD node normalized for $v$, where $v$ is an ancestor of $v_X$ or $v = v_X$. Then, 

$$vr(\alpha) = \sum_y [MC(\alpha|y) \geq T],$$

where $Y = Vars(v) \cap X$.

Proof. The proof is by induction on the distance of $v$ to $v_X$.

Base case: Suppose $v = v_X$. As $v$ is not an ancestor of $v_X$, Line 2 or Line 3 computes $MC(\alpha)$ by Lemma 29. So, Line 5 computes $vr(\alpha) = [MC(\alpha) \geq T]$. Note that $Y = \emptyset$, and so the only possible instantiation of $Y$ is $y = \top$. As $\alpha = \alpha|y$, $vr(\alpha) = \sum_y [MC(\alpha|y) \geq T]$.

Inductive step: Suppose that $v$ is an ancestor of $v_X$ and that the lemma holds for SDDs nodes that are normalized for $v'$. Let $Y^l = Vars(v^l)$ and $Y^r = Vars(v^r) \cap X$. Note that $Y = Y^l \cup Y^r$. Let $\alpha$ be $\{(p_1, s_1), \ldots, (p_n, s_n)\}$. As each $p_i$ is normalized for $v^l$, which is not an ancestor of $v_X$, $vr(p_i) = MC(p_i)$ by Lemma 29. Also, as each $s_i$ is normalized for $v^r$, $vr(s_i) = \sum_{y^r} [MC(s_i|y^r) \geq T]$ by the induction hypothesis. So, Line 3 computes the following:
\begin{align*}
vr(\alpha) &= \sum_i vr(p_i) \times vr(s_i) \\
&= \sum_i MC(p_i) \left( \sum_{y} [MC(s_i|y^r) \geq T] \right) \\
&= \sum_i \left( \sum_{y^l=p_i} 1 \right) \left( \sum_{y^r} [MC(s_i|y^r) \geq T] \right) \\
&= \sum_i \sum_{y^l=p_i, \ y^r} [MC(s_i|y^r) \geq T] \\
&= \sum_{y^l, y^r} [MC(\alpha|y^r) \geq T] \tag{as Y = Y^l \cup Y^r}. \hspace{1cm} \blacksquare
\end{align*}

Since the root of SDD $S$ is normalized for an ancestor of $v_X$, by Lemma 30 we conclude that Algorithm 9 returns $\sum_x [MC(S|x) \geq T]$ on Line 6, which is the same as $MMS(S, X, T)$. Hence, Proposition 10 holds.

8.B Soundness of Algorithm 10

We next prove the soundness of Algorithm 10 whose proof was outlined earlier. We start by showing two lemmas.

**Lemma 36.** Let $\alpha$ be a function over variables $Z$, and let $Y \subseteq Z$. Then $\phi_{\alpha}(y \ e) = \phi_{y} \phi_{\alpha|y}(e)$.

**Proof.** The lemma holds due to the following.

\[
\phi_{\alpha}(y \ e) = \sum_{z=\alpha} \phi_z \sum_{z=y \ e} \phi_y \phi_{z|y} = \phi_y \sum_{x=\alpha} \phi_x = \phi_y \phi_{\alpha|y}(e). \hspace{1cm} \blacksquare
\]

**Lemma 37.** $\phi_{\alpha}(d \mid y \ e) = \phi_{\alpha|y}(d \mid e)$, where $\alpha$ is defined over variables $Z$ and $Y \subseteq Z$.  

Proof. The following holds due to Lemma 36.

\[ \phi_{\alpha}(d \mid y e) = \frac{\phi_{\alpha}(dye)}{\phi_{\alpha}(ye)} = \frac{\phi_y \phi_{\alpha|y}(de)}{\phi_y \phi_{\alpha|y}(e)} = \phi_{\alpha|y}(d \mid e). \]

\[ \square \]

We now prove the lemmas that show Algorithm 10 is sound.

**Lemma 31.** Let \( \alpha \) be an SDD node normalized for vtree \( v \). Then, \( vr_1(\alpha) = \phi_{\alpha}(e_v) \).

**Proof.** \( vr_1(\alpha) \) is computed during the first pass over \( S \) (Lines 1–4). If \( \alpha \) is a terminal SDD, Line 3 clearly computes \( vr_1(\alpha) = \phi_{\alpha}(e_v) \). Suppose \( \alpha \) is an \((X, Y)\)-partition. Let \( \alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\} \), and let \( e_v = e^l e^r \) where \( e^l \) and \( e^r \) are the partial instantiations over \( X \) and \( Y \) respectively. Line 4 then computes \( \sum_i \phi_{p_i}(e^l) \times \phi_{s_i}(e^r) \), which is equal to \( \phi_{\alpha}(e_v) \) by Lemma 35.

**Lemma 32.** Let \( \alpha \) be an SDD node normalized for vtree \( v \), where \( v \) is not an ancestor of \( v_H \). Lines 7–8 compute \( \phi_{\alpha}(d e_v) \) if \( D \) is contained in vtree \( v \), and \( \phi_{\alpha}(e_v) \) otherwise.

**Proof.** Since \( v \) is not an ancestor of \( v_H \), Line 10 will never be executed for the descendants of \( \alpha \). Hence, the computations of Lines 7–8 is analogous to the computation of Lines 1–4, as in Lemma 31, except where we include \( d \) as part of the evidence \( e_v \) (if variable \( D \) is contained in vtree node \( v \)). That is, we compute \( vr_2(\alpha) = \phi_{\alpha}(de_v) \), if \( D \) is contained in \( v \), and \( vr_2(\alpha) = \phi_{\alpha}(e_v) \) otherwise.

**Lemma 33.** Let \( \alpha \) be an SDD node normalized for vtree \( v \), where \( v \) is an ancestor of \( v_H \) or \( v = v_H \). Then,

\[ vr_2(\alpha) = \sum_y [\phi_{\alpha}(d \mid ye) \geq T] \phi_{\alpha}(ye), \]

where \( Y = Vars(v) \cap H \).

**Proof.** The proof is similar to the proof of Lemma 30, and is done by induction on the distance of \( v \) to \( v_H \).
**Base case:** Suppose \( v = v_H \). As \( v \) is not an ancestor of \( v_H \), Line 7 or Line 8 computes \( \phi_\alpha(d,e) \) by Lemma 32. Moreover, \( vr_1(\alpha) = \phi_\alpha(e) \) by Lemma 31. So, Line 10 computes \( vr_2(\alpha) = [\phi_\alpha(d \mid e) \geq T] \phi_\alpha(e) \). Note that \( Y = \emptyset \), and so the only possible instantiation of \( Y \) is \( y = T \). As \( \alpha = \alpha \mid y \), \( vr_2(\alpha) = \sum_y [\phi_\alpha(d \mid ye) \geq T] \phi_\alpha(ye) \).

**Inductive step:** Suppose that \( v \) is an ancestor of \( v_H \) and that the lemma holds for SDD nodes that are normalized for \( v' \). Let \( Y^l = Vars(v^l) \) and \( Y^r = Vars(v^r) \cap H \). Note that \( Y = Y^l \cup Y^r \). Let \( \alpha \) be \( \{(p_1, s_1), \ldots, (p_n, s_n)\} \). Note that each \( p_i \) is normalized for \( v^l \), and neither \( D \) nor \( E \) appears in \( vars(v^l) \). Hence, via Lemma 32, \( vr_2(p_i) = \phi_{p_i}(T) = \phi_{p_i} \). Further, as each \( s_i \) is normalized for \( v^r \), we have \( vr_2(s_i) = \sum_{y^r} [\phi_{s_i}(d \mid y^r e) \geq T] \phi_{s_i}(y^r e) \) by the induction hypothesis. So, Line 8 computes the following (justifications are provided at the end):

\[
vr_2(\alpha) = \sum_i vr_2(p_i) \times vr_2(s_i)
\]

\[
= \sum_i \phi_{p_i} \left( \sum_{y^r} [\phi_{s_i}(d \mid y^r e) \geq T] \phi_{s_i}(y^r e) \right)
\]

\[
= \sum_i \sum_{y^r} \phi_{y^r} \left[ \phi_{s_i}(d \mid y^r e) \geq T \right] \phi_{s_i}(y^r e) \quad (8.8)
\]

\[
= \sum_i \sum_{y^r} \left[ \phi_{\alpha_{y^r}}(d \mid y^r e) \geq T \right] \phi_{\alpha_{y^r}}(y^r e) \quad (8.9)
\]

\[
= \sum_i \sum_{y^r} \left[ \phi_\alpha(d \mid y^r e) \geq T \right] \phi_\alpha(y^r e) \quad (8.10)
\]

\[
= \sum_{y^l, y^r} \left[ \phi_{\alpha}(d \mid y^l y^r e) \geq T \right] \phi_{\alpha}(y^l y^r e) \quad (8.11)
\]

\[
= \sum_y \left[ \phi_{\alpha}(d \mid ye) \geq T \right] \phi_{\alpha}(ye). \quad (8.12)
\]

Equations (8.8), (8.9), and (8.10) are due to Lemma 34, Lemma 36, and Lemma 37, respectively. Equation (8.11) holds as primes are partitions. Equation (8.12) holds as \( Y = Y^l \cup Y^r \).
As the root of SDD $S$ is normalized for an ancestor of $v_H$, by Lemma 33,
\[ vr_2(S) = \sum_h [\phi_S(d \mid h e) \geq T] \phi_S(h e). \]
We also know that $vr_1(S) = \phi_S(e)$. So, Algorithm 10 returns $SDP_S(d, H, e, T)$ on Line 13. Hence, Proposition 12 holds.

8.C Complexity of SDP on Weighted CNFs

We now show that SDP on weighted CNFs is a $PP^P$-complete problem (Theorem 26). In particular, we reduce SDP on a weighted CNF to and from SDP on a Bayesian network, which is $PP^P$-complete. First, we can encode a weighted CNF to a Bayesian network and vice-versa, where the weighted model count of a CNF is equivalent to the probability of evidence in a Bayesian network. We can encode a weighted CNF to a Bayesian network, e.g., as shown by [CXD12] (we additionally encode CNF weights as priors in the network). We can encode a Bayesian network as a weighted CNF, e.g., as shown by [CD08] (using ENC1). The reductions are polynomial-time, and results in a one-to-one correspondence between the weighted CNF models and the (non-zero) rows of the joint distribution induced by the network. Hence, the corresponding SDPs are equivalent.
CHAPTER 9

Conclusion

Knowledge compilation is a well-established area in AI that has been influential in solving computationally hard problems. This area has settled on three major research directions: (1) identifying new tractable representations that are characterized by their succinctness and polytime support for certain queries and transformations; (2) developing efficient knowledge compilers; and (3) using those representations and compilers in various applications, such as diagnosis, planning, and probabilistic reasoning.

In this thesis, we have tackled a number of challenges across all three research directions listed above. In particular, we developed new compilation methods that provide theoretical guarantees on the time and space complexities for various representations. For that, we have introduced new complexity parameters that strictly dominate well-known complexity parameters in knowledge compilation. These developments led to significant improvements in the known size upper bounds for the influential language DNNF and its subsets. Further, by augmenting one of the new compilation methods with advanced techniques from the SAT literature, we have developed a state-of-the-art SDD compiler. This compiler is based on a new representation, called Decision-SDD. We have provided a detailed formal description of the new compiler with the hope that it would facilitate the development of efficient compilers by the community. Moreover, we have introduced a new methodology to compile DNNFs without enforcing determinism, which led to the first knowledge compiler with this property.
In addition to these contributions, we have extended the reach of knowledge compilation in solving Beyond-NP problems. In particular, we have developed methods to efficiently solve problems in the highly intractable complexity class $\text{PP}^{\text{PP}}$. For that, we have identified a new representation which is tractable for $\text{PP}^{\text{PP}}$-complete problems. We believe that the advances made here would help in further establishing knowledge compilation as a general methodology for computation.

Overall, our hope is that this thesis have contributed some new and fundamental insights into solving computationally hard problems using knowledge compilation.
REFERENCES


