RICE UNIVERSITY

Weighted Model Counting with Algebraic Decision Diagrams

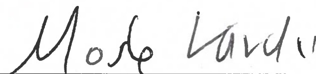
by

Vu Hoang Nguyen Phan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE

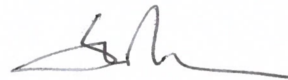
Master of Science

APPROVED, THESIS COMMITTEE:



Moshe Vardi, Ph.D., Chairperson
University Professor,
Karen Ostrum George
Distinguished Service Professor
in Computational Engineering,
Professor in Computer Science

Devika Subramanian, Ph.D.
Professor in Computer Science,
Professor in Electrical Engineering



Swarat Chaudhuri, Ph.D.
Associate Professor in Computer Science

Houston, Texas

December, 2019

ABSTRACT

Weighted Model Counting

with Algebraic Decision Diagrams

by

Vu Hoang Nguyen Phan

We present an algorithm to compute exact literal-weighted model counts of Boolean formulas in conjunctive normal form. Our algorithm employs dynamic programming and uses algebraic decision diagrams as the primary data structure. We implement this algorithm to create `ADDMC`, a new model counter. We empirically evaluate various heuristics that can be used with `ADDMC`. We then compare `ADDMC` to four state-of-the-art exact weighted model counters (`Cachet`, `c2d`, `d4`, and `miniC2D`) on 1914 standard model counting benchmarks and show that `ADDMC` significantly improves the virtual best solver.

Acknowledgments

I appreciate Dr. Moshe Vardi's guidance as my research advisor and the chairperson of my M.S. committee. I am grateful to Dr. Devika Subramanian and Dr. Swarat Chaudhuri for serving on the committee and evaluating my thesis. I acknowledge the support of my family and friends. In particular, Jeffrey Dudek coauthored my first paper with Dr. Vardi [Dudek et al., 2020]; my thesis is based on this paper. In addition, I appreciate feedback from Dror Fried, Aditya Shrotri, Lucas Tabajara, and others. Also, I thank Dr. Jan Odegard for helping with my research computing and for making this thesis template.

This work was supported in part by the NSF (grants CNS-1338099, CCF-1704883, IIS-1527668, IIS-1830549, and DMS-1547433) as well as by Rice University.

Contents

Abstract ii

Acknowledgments iii

List of Figures **vii**

List of Tables **viii**

List of Equations **ix**

List of Algorithms **x**

1 Introduction **1**

2 Preliminaries **5**

2.1 Weighted Model Counting 5

2.2 Algebraic Decision Diagrams 6

3 Using Algebraic Decision Diagrams for Weighted Model Counting **9**

3.1 General Weighted Model Counting 9

3.2 Literal-Weighted Model Counting of Conjunctive Normal Form Formulas 13

4 Dynamic Programming for Weighted Model Counting 16

4.1 Proof of Theorem 3 19

4.2 Heuristics for `get-diagram-var-order` and `get-cluster-var-order` 22

4.3 Heuristics for `get-clause-rank` 24

4.4 Heuristics for `choose-cluster` 25

4.5 Similar Techniques in Model Counting and Related Problems 26

4.5.1 Probabilistic Inference 26

4.5.2 Constraint Satisfaction 27

5 Empirical Evaluation 29

5.1 Experiment 1A: Comparing All `ADDMC` Heuristic Configurations 30

5.1.1 Correctness Analysis 31

5.1.2 Performance Analysis 31

5.2 Experiment 1B: Comparing Best `ADDMC` Heuristic Configurations . . . 33

5.2.1 Correctness Analysis 34

5.2.2 Performance Analysis 34

5.3 Experiment 2: Comparing `ADDMC` to Weighted Model Counters 36

5.3.1 Correctness Analysis 37

5.3.2 Performance Analysis 37

5.3.3 Predicting ADDMC Performance 40

5.4 Experiment 3: Comparing ADDMC to Bayesian Inference Engines . . . 43

6 Discussion **45**

Bibliography **48**

List of Figures

1 The graph G of an ADD with variable set $X = \{x_1, x_2, x_3\}$, carrier set $S = \mathbb{R}$, and diagram variable order $\pi(x_i) = i$ for $i = 1, 2, 3$. 7

2 Experiment 1A: a cactus plot of the numbers of benchmarks solved by the best, second-best, median, and worst `ADDMC` heuristic configurations. 32

3 Experiment 1B: a cactus plot of the numbers of benchmarks solved in 1000 seconds by the five best `ADDMC` heuristic configurations in Experiment 1A (10 seconds) and their virtual best solver (*VBS5*). . . 35

4 Experiment 2: a cactus plot of the numbers of benchmarks solved by five weighted model counters and two virtual best solvers (`VBS1` with `ADDMC` and `VBS0` without `ADDMC`). 39

5 A cactus plot of the number of benchmarks, in total and solved by `ADDMC`, for various upper bounds for MAVCs. The MAVCs of the 1404 benchmarks solved by `ADDMC` within 1000 seconds range from 4 to 246. 41

6 A scatter plot of the solving time of `ADDMC` against the MAVC for each of the 1404 benchmarks solved by `ADDMC` within 1000 seconds. . 42

List of Tables

1 Experiment 1A: the numbers of benchmarks solved (of 1914) in 10 seconds by the best (1st-5th), median, and worst `ADDMC` heuristic configurations. 33

2 Experiment 1B: the numbers of benchmarks solved (of 1914) in 1000 seconds by the five best `ADDMC` heuristic configurations in Experiment 1A (10 seconds) and their virtual best solver (*VBS5*). 36

3 Experiment 2: the numbers of benchmarks solved (of 1914) in 1000 seconds — uniquely (i.e., benchmarks solved by no other solver), fastest, and in total — by five weighted model counters and two virtual best solvers (`VBS1` and `VBS0`). 38

List of Equations

1 Equation (1): CNF literal-weighted model count 15

3 Equation (3): Floating-point equality tolerance 31

List of Algorithms

1 CNF literal-weighted model counting with ADDs 17

Chapter 1

Introduction

Model counting is a fundamental problem in artificial intelligence, with applications in machine learning, probabilistic reasoning, and verification [Domshlak and Hoffmann, 2007, Biere et al., 2009, Naveh et al., 2007]. Given an input set of constraints, with the focus in this work on Boolean constraints, the model counting problem is to count the number of satisfying assignments. Although this problem is #P-Complete [Valiant, 1979], a variety of tools exist that can handle industrial sets of constraints, cf. [Sang et al., 2004, Oztok and Darwiche, 2015].

Dynamic programming is a powerful technique that has been applied across computer science [Howard, 1966], including to model counting [Bacchus et al., 2009, Samer and Szeider, 2010]. The key idea is to solve a large problem by solving a sequence of smaller subproblems and then incrementally combining these solutions into the final result. Dynamic programming provides a natural framework to solve a variety of problems defined on sets of constraints: subproblems can be formed by partitioning the constraints into sets, called *clusters*. This framework has also been instantiated into algorithms for database-query optimization [McMahan et al., 2004] and SAT-solving [Uribe and Stickel, 1994, Aguirre and Vardi, 2001, Pan and Vardi, 2004]. Techniques for local computation can also be seen as a variant of this framework, e.g., in theorem

proving [Wilson and Mengin, 1999] or probabilistic inference [Shenoy and Shafer, 2008].

In this work, we study two algorithms that follow this dynamic-programming framework and can be adapted for model counting: *bucket elimination* [Dechter, 1999] and *Bouquet's Method* [Bouquet, 1999]. Bucket elimination aims to minimize the amount of information needed to be carried between subproblems. When this information must be stored in an uncompressed table, bucket elimination will, with some carefully chosen sequence of clusters, require the minimum possible amount of intermediate data (as governed by the treewidth of the input formula [Bacchus et al., 2009]). Intermediate data, however, need not be stored uncompressed. Several works have shown that using compact representations of intermediate data can dramatically improve bucket elimination for Bayesian inference [Poole and Zhang, 2003, Sanner and McAllester, 2005, Chavira and Darwiche, 2007]. Moreover, it has been observed that using compact representations – in particular, *binary decision diagrams (BDDs)* – can allow Bouquet's Method to outperform bucket elimination for SAT-solving [Pan and Vardi, 2004]. Compact representations are therefore promising to improve existing dynamic-programming-based algorithms for model counting [Bacchus et al., 2009, Samer and Szeider, 2010].

In particular, we consider the use of *algebraic decision diagrams* (*ADDs*) [Bahar et al., 1997] for model counting in a dynamic-programming framework. An ADD is a compact representation of a real-valued function as a directed acyclic graph.

For functions with logical structure, the ADD representation can be exponentially smaller than the explicit representation. ADDs have been successfully used as part of dynamic-programming frameworks for Bayesian inference [Chavira and Darwiche, 2007, Gogate and Domingos, 2012] and stochastic planning [Hoey et al., 1999]. Although ADDs have been used for model counting outside of a dynamic-programming framework [Fargier et al., 2014], no prior work uses ADDs for model counting as part of a dynamic-programming framework.

The construction and resultant size of an ADD depend heavily on the choice of an order on the variables of the ADD, called a *diagram variable order*. Some variable orders may produce ADDs that are exponentially smaller than others for the same real-valued function. A variety of techniques exist in prior work to heuristically find diagram variable orders [Tarjan and Yannakakis, 1984, Koster et al., 2001]. In addition to the diagram variable order, both bucket elimination and Bouquet's Method require another order on the variables to build and arrange the clusters of input constraints; we call this a *cluster variable order*. We show that the choice of heuristics to find cluster variable orders has a significant impact on the runtime performance of both bucket elimination and Bouquet's Method.

The primary contribution of this work is a dynamic-programming framework for weighted model counting that utilizes ADDs as a compact data structure. In particular:

1. We lift the BDD-based approach for Boolean satisfiability of [Pan and Vardi,

2004] to an ADD-based approach for weighted model counting.

2. We implement this algorithm using ADDs and a variety of existing heuristics to produce `ADDMC`, a new weighted model counter.

3. We perform an experimental comparison of these heuristic techniques in the context of weighted model counting.

4. We perform an experimental comparison of `ADDMC` to four state-of-the-art weighted model counters (`Cachet`, `c2d`, `d4`, and `miniC2D`) and show that `ADDMC` improves the virtual best solver on 763 benchmarks (of 1914 in total).

In Chapter 2, we formally define weighted model counting and algebraic decision diagrams. In Chapter 3, we outline the theoretical foundation for performing weighted model counting with ADDs. In Chapter 4, we present an algorithm for performing weighted model counting through dynamic-programming techniques, and discuss a variety of existing heuristics that can be used in the algorithm. In Chapter 5, we compare the performance of various heuristics in `ADDMC` and demonstrate that Bouquet's Method is competitive with bucket elimination. Also, we compare `ADDMC` against four state-of-the-art tools (`Cachet`, `c2d`, `d4`, and `miniC2D`) on 1914 standard model counting benchmarks. Finally, we conclude in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we introduce weighted model counting, the central problem of this work, and algebraic decision diagrams, the primary data structure we use to solve weighted model counting.

2.1 Weighted Model Counting

The central problem of this work is to compute the weighted model count of a Boolean formula, which we now define.

Definition 1. Let $\varphi : 2^X \to \{0, 1\}$ be a Boolean function over a set X of variables, and let $W : 2^X \to \mathbb{R}$ be an arbitrary function. The *weighted model count* of φ w.r.t. W is

$$W(\varphi) = \sum_{\tau \in 2^X} \varphi(\tau) \cdot W(\tau).$$

The function $W : 2^X \to \mathbb{R}$ is called a *weight function*. In this work, we focus on so-called *literal-weight functions*, where the weight of a model can be expressed as the product of weights associated with all satisfied literals. That is, where the weight function W can be expressed, for all $\tau \in 2^X$, as

$$W(\tau) = \prod_{x \in \tau} W^+(x) \cdot \prod_{x \in X \setminus \tau} W^-(x)$$

for some functions $W^+(x), W^-(x) : X \to \mathbb{R}$. One can interpret these literal-weight functions W as assigning a real number weight to each literal: $W^+(x)$ to x and $W^-(x)$ to $\neg x$. It is common to restrict attention to weight functions whose range is $\mathbb{R}$ or just the interval $[0, 1]$.

When the formula φ is given in *conjunctive normal form (CNF)*, computing the literal-weighted model count is #P-Complete [Valiant, 1979]. Several algorithms and tools for weighted model counting directly reason about the CNF representation. For example, `Cachet` uses DPLL search combined with component caching and clause learning to perform weighted model counting [Sang et al., 2004].

If φ is given in a compact representation (e.g., as a binary decision diagram (BDD) [Bryant, 1986] or as a sentential decision diagram (SDD) [Darwiche, 2011]) computing the literal-weighted model count can be done in time polynomial in the size of the representation. One recent tool for weighted model counting that exploits this is `miniC2D`, which compiles the input CNF formula into an SDD and then performs a polynomial-time count on the SDD [Oztok and Darwiche, 2015]. Note that these compact representations may still be exponential in the size of the corresponding CNF formula in the worst case.

2.2 Algebraic Decision Diagrams

The central data structure we use in this work is *algebraic decision diagram* (*ADD*) [Bahar et al., 1997], a compact representation of a function as a directed acyclic graph.

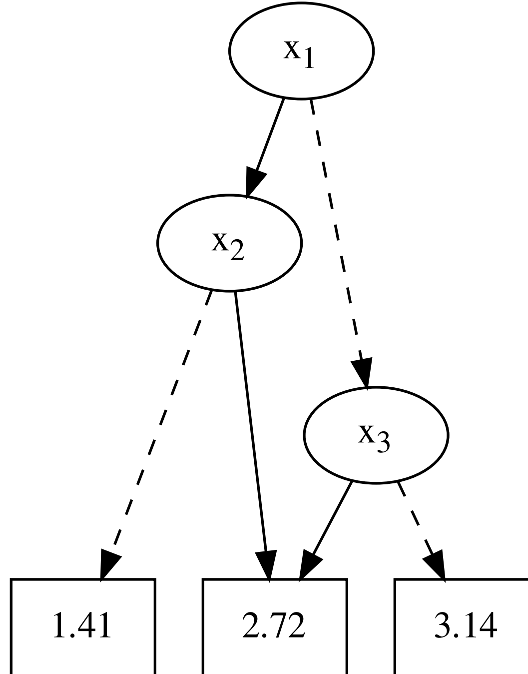


Figure 1 : The graph G of an ADD with variable set $X = \{x_1, x_2, x_3\}$, carrier set $S = \mathbb{R}$, and diagram variable order $\pi(x_i) = i$ for $i = 1, 2, 3$.

Formally, an ADD is a tuple (X, S, π, G), where X is a set of Boolean variables, S is an arbitrary set (called the *carrier set*), $\pi : X \to \mathbb{Z}^+$ is an injection (called the *diagram variable order*), and G is a rooted directed acyclic graph satisfying the following three properties. First, every terminal node of G is labeled with an element of S. Second, every non-terminal node of G is labeled with an element of X and has two outgoing edges labeled 0 and 1. Finally, for every path in G, the labels of the visited non-terminal nodes must occur in increasing order under π.

Figure 1 is a graphical example of an ADD. In this figure, if a directed edge from an oval node is solid (respectively dashed), then corresponding Boolean variable is assigned 1 (respectively 0).

ADDs were originally designed for matrix multiplication and shortest path algorithms

[Bahar et al., 1997]. ADDs have also been used for stochastic model checking [Kwiatkowska et al., 2007] and stochastic planning [Hoey et al., 1999]. In this work, we do not need arbitrary carrier sets; it is sufficient to consider ADDs with $S = \mathbb{R}$.

An ADD (X, S, π, G) is a compact representation of a function $f : 2^X \to S$. Although there are many ADDs representing each such function f, for each injection $\pi : X \to \mathbb{Z}^+$, there is a unique minimal ADD that represents f with π as the diagram variable order, called the *canonical ADD*. ADDs can be minimized in polynomial time, so it is typical to only work with canonical ADDs. Given two ADDs representing functions f and g, the ADDs representing $f + g$ and $f \cdot g$ can also be computed in polynomial time.

The choice of diagram variable order can have a dramatic impact on the size of the ADD. A variety of techniques exist to heuristically find diagram variable orders. Moreover, since binary decision diagrams (BDDs) [Bryant, 1986] can be seen as ADDs with carrier set $S = \{0, 1\}$, there is significant overlap with the techniques to find variable orders for BDDs. We discuss these heuristics in more detail in Chapter 4.

Several packages exist for efficiently manipulating ADDs. Here we use the package `CUDD` [Somenzi, 2015], which supports carrier sets $S = \{0, 1\}$ and (using floating-point arithmetic) $S = \mathbb{R}$. `CUDD` implements several ADD operations, such as addition, multiplication, and projection.

Chapter 3

Using Algebraic Decision Diagrams for Weighted Model Counting

An ADD with carrier set $\mathbb{R}$ can be used to represent both a Boolean function $\varphi : 2^X \to \{0, 1\}$ and a weight function $W : 2^X \to \mathbb{R}$. ADDs are thus a natural candidate as a data structure for weighted model counting algorithms.

In this chapter, we outline theoretical foundations for performing weighted model counting with ADDs. We consider first the general case of weighted model counting. We then specialize to literal-weighted model counting of CNF formulas and show how the technique of early projection can take advantage of such factored representations of Boolean formulas φ and weight functions W.

3.1 General Weighted Model Counting

We assume that the Boolean formula φ and the weight function W are represented as ADDs. The goal is to compute $W(\varphi)$, the weighted model count of φ w.r.t. W. To do this, we define two operations on functions $2^X \to \mathbb{R}$ that can be efficiently computed using the ADD representation: *product* and *projection*. These operations are combined in Theorem 1 to perform weighted model counting.

First, we define the *product* of two functions.

Definition 2. Let X and Y be sets of variables. The *product* of functions $A : 2^X \to \mathbb{R}$ and $B : 2^Y \to \mathbb{R}$ is the function $A \cdot B : 2^{X \cup Y} \to \mathbb{R}$ defined for all $\tau \in 2^{X \cup Y}$ by

$$(A \cdot B)(\tau) = A(\tau \cap X) \cdot B(\tau \cap Y).$$

Note that the operator $\cdot$ is commutative and associative, and it has the identity element $\mathbf{1} : 2^{\varnothing} \to \mathbb{R}$ (that maps $\varnothing$ to 1). If $\varphi : 2^X \to \{0,1\}$ and $\psi : 2^Y \to \{0,1\}$ are Boolean formulas, their product $\varphi \cdot \psi$ is the Boolean function corresponding to their conjunction $\varphi \wedge \psi$.

Second, we define the *projection* of a Boolean variable x in a real-valued function A, which reduces the number of variables in A by "summing out" x. Projection in real-valued functions is similar to *variable elimination* in Bayesian networks [Zhang and Poole, 1994]. (Variable elimination "sums out" a variable v from a set S of potentials by: removing all potentials p_i containing v from S, computing a new potential $p' = \sum_{v \in \{0,1\}} \prod_i p_i$, and adding p' to S.)

Definition 3. Let X be a set of variables and $x \in X$. The *projection* of $A : 2^X \to \mathbb{R}$ w.r.t. x is the function $\exists_x A : 2^{X \setminus \{x\}} \to \mathbb{R}$ defined for all $\tau \in 2^{X \setminus \{x\}}$ by

$$(\exists_x A)(\tau) = A(\tau) + A(\tau \cup \{x\}).$$

One can check that projection is commutative, i.e., $\exists_x \exists_y A = \exists_y \exists_x A$ for all variables $x, y \in X$ and functions $A : 2^X \to \mathbb{R}$. If $X = \{x_1, x_2, \ldots, x_n\}$, define

$$\exists_X A = \exists_{x_1} \exists_{x_2} \ldots \exists_{x_n} A.$$

We are now ready to use product and projection to do weighted model counting, through the following theorem.

Theorem 1. Let $\varphi : 2^X \to \{0,1\}$ be a Boolean function over a set X of variables, and let $W : 2^X \to \mathbb{R}$ be an arbitrary weight function. Then

$$W(\varphi) = (\exists_X(\varphi \cdot W))(\varnothing).$$

Proof. Assume the variables in X are $x_1, x_2, \ldots, x_n$. Now, for an arbitrary function $A : 2^X \to \mathbb{R}$, we have:

$$\begin{aligned}
\sum_{\tau \in 2^X} A(\tau) &= \sum_{\tau \in 2^{X \setminus \{x_n\}}} (A(\tau) + A(\tau \cup \{x_n\})) && \text{(regrouping terms)} \\
&= \sum_{\tau \in 2^{X \setminus \{x_n\}}} ((\exists_{x_n} A)(\tau)) && \text{(Definition 3)}
\end{aligned}$$

Similarly, projecting all variables in X:

$$\begin{aligned}
\sum_{\tau \in 2^X} A(\tau) &= \sum_{\tau \in 2^{X \setminus \{x_n\}}} (\exists_{x_n} A)(\tau) \\
&= \sum_{\tau \in 2^{X \setminus \{x_{n-1}, x_n\}}} (\exists_{x_{n-1}} \exists_{x_n} A)(\tau) \\
&\vdots \\
&= \sum_{\tau \in 2^{\varnothing}} (\exists_{x_1} \dots \exists_{x_{n-1}} \exists_{x_n} A)(\tau) \\
&= (\exists_{x_1} \dots \exists_{x_{n-1}} \exists_{x_n} A)(\varnothing) \\
&= (\exists_X A)(\varnothing)
\end{aligned}$$

When A is the specific function $\varphi \cdot W : 2^X \to \mathbb{R}$, we have:

$$\sum_{\tau \in 2^X} (\varphi \cdot W)(\tau) = (\exists_X (\varphi \cdot W))(\varnothing)$$

Finally:

$$\begin{aligned}
(\exists_X (\varphi \cdot W))(\varnothing) &= \sum_{\tau \in 2^X} (\varphi \cdot W)(\tau) && \text{(as above)} \\
&= \sum_{\tau \in 2^X} \varphi(\tau) \cdot W(\tau) && \text{(Definition 2)} \\
&= W(\varphi) && \text{(Definition 1)}
\end{aligned}$$

□

Theorem 1 suggests that $W(\varphi)$ can be computed by constructing an ADD for φ and another for W, computing the ADD for their product $\varphi \cdot W$, and performing a

sequence of projections to obtain the final weighted model count. Unfortunately, this "monolithic" approach is infeasible in most interesting cases: the ADD representation of $\varphi \cdot W$ is often too large, even with the best possible diagram variable order.

Instead, we next show a technique for avoiding the construction of an ADD for $\varphi \cdot W$ by rearranging the products and projections.

3.2 Literal-Weighted Model Counting of Conjunctive Normal Form Formulas

A key technique in symbolic computation is *early projection*: when performing a product followed by a projection (as in Theorem 1), it is sometimes possible and advantageous to perform the projection first. Early projection is possible when one component of the product does not depend on the projected variable. Early projection has been used in a variety of settings, including database-query optimization [Kolaitis and Vardi, 2000], symbolic model checking [Burch et al., 1991], and satisfiability solving [Pan and Vardi, 2005]. The formal statement is as follows.

Theorem 2 (Early Projection)**.** Let X and Y be sets of variables, $A : 2^X \to \mathbb{R}$, and $B : 2^Y \to \mathbb{R}$. For all $x \in X \setminus Y$,

$$\exists_x (A \cdot B) = (\exists_x A) \cdot B.$$

As a corollary, for all $X' \subseteq X \setminus Y$,

$$\exists_{X'} (A \cdot B) = (\exists_{X'} A) \cdot B.$$

Proof. For every $\tau \in 2^{(X \cup Y) \setminus \{x\}}$, we have:

$$
\begin{aligned}
&(\exists_x (A \cdot B))(\tau) \\
&\quad = (A \cdot B)(\tau) + (A \cdot B)(\tau \cup \{x\}) && \text{(Definition 3)} \\
&\quad = A(\tau \cap X) \cdot B(\tau \cap Y) + A((\tau \cup \{x\}) \cap X) \cdot B((\tau \cup \{x\}) \cap Y) && \text{(Definition 2)} \\
&\quad = A(\tau \cap X) \cdot B(\tau \cap Y) + A((\tau \cup \{x\}) \cap X) \cdot B(\tau \cap Y) && \text{(as } x \notin Y\text{)} \\
&\quad = A(\tau \cap X) \cdot B(\tau \cap Y) + A(\tau \cap X \cup \{x\}) \cdot B(\tau \cap Y) && \text{(as } x \in X\text{)} \\
&\quad = (A(\tau \cap X) + A(\tau \cap X \cup \{x\})) \cdot B(\tau \cap Y) && \text{(common factor)} \\
&\quad = (\exists_x A)(\tau \cap X) \cdot B(\tau \cap Y) && \text{(Definition 3)} \\
&\quad = (\exists_x A)(\tau \cap (X \setminus \{x\})) \cdot B(\tau \cap Y) && \text{(as } x \notin \tau\text{)} \\
&\quad = ((\exists_x A) \cdot B)(\tau) && \text{(Definition 2)}
\end{aligned}
$$

□

The use of early projection in Theorem 1 is quite limited when φ and W have already been represented as ADDs, since on many benchmarks both φ and W depend on most of the variables. If φ is a CNF formula and W is a literal-weight function, however, both φ and W can be rewritten as products of smaller functions. This can significantly increase the applicability of early projection.

Assume that φ is a CNF formula, i.e., given as a set of clauses. For every clause $\gamma \in \varphi$, observe that γ is a Boolean function $\gamma : 2^{X_\gamma} \to \{0, 1\}$ (where $X_\gamma \subseteq X$ is the set of variables appearing in γ) that maps satisfying assignments to 1 and unsatisfying assignments to 0. One can check using Definition 2 that $\varphi = \prod_{\gamma \in \varphi} \gamma$.

Similarly, assume that $W : 2^X \to \mathbb{R}$ is a literal-weight function. For every $x \in X$, define $W_x : 2^{\{x\}} \to \mathbb{R}$ to be the function that maps $\varnothing$ to $W^-(x)$ and $\{x\}$ to $W^+(x)$. One can check using Definition 2 that $W = \prod_{x \in X} W_x$.

When φ is a CNF formula and W is a literal-weight function, we can rewrite Theorem 1 as

$$W(\varphi) = \left(\exists_X \left(\prod_{\gamma \in \varphi} \gamma \cdot \prod_{x \in X} W_x \right) \right) (\varnothing). \tag{1}$$

By taking advantage of the associative and commutative properties of multiplication as well as the commutative property of projection, it is possible to rearrange Equation 1 in order to apply early projection. We present an algorithm to perform this rearrangement in the following chapter.

Chapter 4

Dynamic Programming for Weighted Model Counting

We now discuss an algorithm for performing literal-weighted model counting of CNF formulas using ADDs through dynamic-programming techniques.

Our algorithm is presented as Algorithm 1. Broadly, our algorithm partitions the clauses of a formula φ into clusters. For each cluster, we construct an ADD corresponding to the conjunction of its clauses. These ADDs are then incrementally combined via the multiplication operator. Throughout, each variable of the ADDs is projected as early as Theorem 2 allows (X_i is the set of variables that can be projected in each iteration i of the second loop). At the end of the algorithm, all variables have been projected, and the resulting ADD has a single node representing the weighted model count. This algorithm can be seen as rearranging the terms of Equation 1 (according to the clusters) in order to perform the projections indicated by X_i at each step i.

The function `get-clause-ADD`(γ, π) constructs the ADD representing the clause γ, using π as the diagram variable order. The remaining functions that appear in Algorithm 1, namely `get-diagram-var-order`, `get-cluster-var-order`, `get-clause-rank`, and `choose-cluster`, represent heuristics that can be used to tune the specifics of

Algorithm 1: CNF literal-weighted model counting with ADDs

Input: X: set of Boolean variables

Input: φ: nonempty CNF formula over X

Input: W: literal-weight function over X

Output: $W(\varphi)$: weighted model count of φ w.r.t. W

```
1  π ← get-diagram-var-order(φ)                      /* injection π : X → ℤ⁺ */
2  ρ ← get-cluster-var-order(φ)                      /* injection ρ : X → ℤ⁺ */
3  m ← max_{x∈X} ρ(x)
4  for i = m, m − 1, ..., 1
5  |   Γ_i ← {γ ∈ φ : get-clause-rank(γ, ρ) = i}     /* collecting clauses γ with
   |      rank i */
6  |   κ_i ← {get-clause-ADD(γ, π) : γ ∈ Γ_i}        /* cluster κ_i contains ADDs of
   |      clauses γ with rank i */
7  |   X_i ← Vars(κ_i) \ ⋃_{p=i+1}^{m} Vars(κ_p)     /* variables already placed in X_i will
   |      not be placed in X_1, X_2, ..., X_{i−1} */
8  for i = 1, 2, ..., m
9  |   if κ_i ≠ ∅
10 |   |   A_i ← ∏_{D∈κ_i} D                         /* product of all ADDs D in cluster κ_i */
11 |   |   for x ∈ X_i
12 |   |   |   A_i ← ∃_x (A_i · W_x)                 /* W_x : 2^{x} → ℝ, represented by an ADD */
13 |   |   if i < m
14 |   |   |   j ← choose-cluster(A_i, i)            /* i < j ≤ m */
15 |   |   |   κ_j ← κ_j ∪ {A_i}
16 return A_m(∅)                                     /* A_m : 2^∅ → ℝ */
```

the algorithm.

We assert the correctness of Algorithm 1 in the following theorem.

Theorem 3. Let X be a set of variables, φ be a nonempty CNF formula over X, and W be a literal-weight function over X. Assume that `get-diagram-var-order` and `get-cluster-var-order` return injections $X \rightarrow \mathbb{Z}^+$. Also assume that all `get-clause-rank` and `choose-cluster` calls satisfy the following conditions:

1. $1 \leq \texttt{get-clause-rank}(\gamma, \rho) \leq m$,
2. $i < \texttt{choose-cluster}(A_i, i) \leq m$, and
3. $X_s \cap \texttt{Vars}(A_i) = \varnothing$ for all integers s such that $i < s < \texttt{choose-cluster}(A_i, i)$.

Then Algorithm 1 returns $W(\varphi)$.

Before giving a full proof of Theorem 3 in Section 4.1, we give a proof sketch here. By Condition 1, we know the set $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ forms a partition of the clauses in φ. Condition 2 ensures that lines 14-15 place A_i in a cluster that has not yet been processed. Also on lines 14-15, Condition 3 prevents A_i from skipping a cluster κ_s which shares some variable y with A_i, as y will be projected at step s. These three invariants are sufficient to prove that Algorithm 1 indeed computes the weighted model count of φ w.r.t. W. All heuristics we describe in this work satisfy the conditions of Theorem 3.

4.1 Proof of Theorem 3

In order to prove Theorem 3, we first state and prove two invariants that hold during the loop at line 8 of Algorithm 1.

First, we prove in the following lemma that the variables in X_i never appear in the clusters κ_j for every $i < j$.

Lemma 1. Assume the conditions of Theorem 3. At every step of the loop at line 8 of Algorithm 1 and for every $1 \leq i < j \leq m$, $X_i \cap \texttt{Vars}(\kappa_j) = \varnothing$.

Proof. We prove this invariant by induction on the steps of the algorithm. The base case (i.e., immediately before line 8) follows from the initial construction of X_i.

During the loop, the only potential problem is at line 15. In particular, consider an iteration $i < m$ where some ADD A_i is added to κ_j (where $j = \texttt{choose-cluster}(A_i, i)$) and assume that the invariant holds before line 15. To prove that the invariant still holds after line 15, consider some $1 \leq s < j$. We prove by cases that $X_s \cap \texttt{Vars}(A_i) = \varnothing$:

- **Case** $s < i$**.** By the inductive hypothesis, we have $X_s \cap \texttt{Vars}(\kappa_i) = \varnothing$. Since $\texttt{Vars}(A_i) \subseteq \texttt{Vars}(\kappa_i)$, it follows that $X_s \cap \texttt{Vars}(A_i) = \varnothing$.

- **Case** $s = i$**.** All variables in X_i are projected from A_i during the loop at line 11. Thus $X_i \cap \texttt{Vars}(A_i) = \varnothing$.

- **Case** $s > i$**.** Since $s < j$, it follows from Condition 3 of Theorem 3 that $X_s \cap \texttt{Vars}(A_i) = \varnothing$.

By the inductive hypothesis, $X_i \cap \texttt{Vars}(B) = \varnothing$ for all other $B \in \kappa_j$. Hence $X_i \cap \texttt{Vars}(\kappa_j) = \varnothing$. $\square$

Next, we use this invariant prove that the ADDs in $\bigcup_{j \geq i} \kappa_j$ always contain sufficient information to compute the weighted model count at iteration i.

Lemma 2. Assume the conditions of Theorem 3 and let $Y_i = \bigcup_{j \geq i} X_i$. At the start of every iteration i of the loop at line 8 of Algorithm 1,

$$W(\varphi) = \left(\exists_{Y_i} \left(\prod_{\substack{j \geq i \\ B \in \kappa_j}} B \cdot \prod_{x \in Y_i} W_x \right) \right) (\varnothing). \tag{2}$$

Proof. We prove this invariant by induction on i.

We first consider iteration $i = 1$. It follows from Condition 1 of Theorem 3 that $\bigcup_{j \geq 1} \Gamma_j = \varphi$. Thus $Y_1 = \texttt{Vars}(\varphi) = X$ and moreover $\prod_{j \geq 1} \prod_{B \in \kappa_j} B = \prod_{\gamma \in \varphi} \texttt{get-clause-ADD}(\gamma, \pi)$. Equation 2 therefore follows directly from Theorem 1.

Next, assume that Equation 2 holds at the start of some iteration $i < m$ and consider Equation 2 at the start of iteration $i + 1$. For convenience, let κ_j refer to its value at the start of iteration i and let κ'_j refer to the value of κ_j at the start of iteration $i + 1$ (for all $j \geq i$).

If $\kappa_i = \varnothing$, then κ_i does not contribute to Equation 2, so Equation 2 remains unchanged (and thus still holds) at the start of iteration $i + 1$. If $\kappa_i \neq \varnothing$, then after lines 10-12 Algorithm 1 computes $A_i = \exists_{X_i} \left(\prod_{D \in \kappa_i} D \cdot \prod_{x \in X_i} W_x \right)$ (using Theorem 2 to rearrange terms). By Condition 2 of Theorem 3, A_i is then placed in κ'_j for some

$i < j \leq m$. Therefore, at the start of iteration $i+1$ we have

$$\exists_{Y_{i+1}} \left(\prod_{\substack{j \geq i+1 \\ B \in \kappa'_j}} B \cdot \prod_{x \in Y_{i+1}} W_x \right)$$
$$= \exists_{Y_{i+1}} \left(A_i \cdot \prod_{\substack{j \geq i+1 \\ B \in \kappa_j}} B \cdot \prod_{x \in Y_{i+1}} W_x \right).$$

Plugging in the value of A_i, this is equal to

$$\exists_{Y_{i+1}} \left(\left(\exists_{X_i} \prod_{D \in \kappa_i} D \cdot \prod_{x \in X_i} W_x \right) \cdot \prod_{\substack{j \geq i+1 \\ B \in \kappa_j}} B \cdot \prod_{x \in Y_{i+1}} W_x \right).$$

Notice Y_i is the disjoint union of Y_{i+1} and X_i. Thus $X_i \cap \texttt{Vars}(W_x) = \varnothing$ for all $x \in Y_{i+1}$. Moreover, by Lemma 1 $X_i \cap \texttt{Vars}\left(\prod_{j \geq i+1} \prod_{B \in \kappa_j} B\right) = \varnothing$. It thus follows from Theorem 2 that

$$\exists_{Y_{i+1}} \left(\prod_{\substack{j \geq i+1 \\ B \in \kappa'_j}} B \cdot \prod_{x \in Y_{i+1}} W_x \right)$$
$$= \exists_{Y_{i+1}} \left(\exists_{X_i} \prod_{\substack{j \geq i \\ B \in \kappa_j}} B \cdot \prod_{x \in Y_i} W_x \right).$$

By the inductive hypothesis, this ADD is exactly $W(\varphi)$ when evaluated at $\varnothing$. It follows that Equation 3 holds at the start of iteration $i+1$ as well. □

Given this second invariant, the proof of Theorem 3 is straightforward.

Proof. By Lemma 2, at the start of iteration m we know that

$$W(\varphi) = \left(\exists_{X_m} \left(\prod_{B \in \kappa_m} B \cdot \prod_{x \in Y_m} W_x \right) \right) (\varnothing).$$

Since

$$A_m = \exists_{X_m} \left(\prod_{B \in \kappa_m} B \cdot \prod_{x \in Y_m} W_x \right), \qquad \text{(using Theorem 2 to rearrange terms)}$$

it follows that line 16 returns $W(\varphi)$. □

In the remainder of this chapter, we discuss various existing heuristics that can be used within Algorithm 1 to implement `get-diagram-var-order` (`get-cluster-var-order`, `get-clause-rank`, and `choose-cluster`) as well as how our techniques are adapted from other problem domains.

4.2 Heuristics for `get-diagram-var-order` and `get-cluster-var-order`

The heuristic chosen for `get-diagram-var-order` indicates the diagram variable order to use in every ADD constructed by Algorithm 1. The heuristic chosen for `get-cluster-var-order` indicates the variable order which, when combined with the heuristic for `get-clause-rank`, is used to order the clauses of φ. (**BE** orders clauses by the smallest variable that appears in each clause, while **BM** orders clauses by the largest variable.) In this work, we consider seven possible heuristics for each variable order: **Random**, **MCS**, **LexP**, **LexM**, **InvMCS**, **InvLexP**, and **InvLexM**.

A simple heuristic for `get-diagram-var-order` and `get-cluster-var-order` is to randomly order the variables, i.e., for a formula over some set X of variables, sample an injection $X \to \{1, 2, \ldots, |X|\}$ uniformly at random. We call this the **Random** heuristic. **Random** is a baseline for comparison of the other variable order heuristics.

For the remaining heuristics, we must define the *Gaifman graph* G_φ of a formula φ. The Gaifman graph of φ has a vertex for every variable in φ. Two vertices are connected by an edge if and only if the corresponding variables appear in the same clause of φ.

One such heuristic is called *Maximum-Cardinality Search* [Tarjan and Yannakakis, 1984]. At each step of the heuristic, the next variable chosen is the variable adjacent in G_φ to the greatest number of previously chosen variables (breaking ties arbitrarily). We call this the **MCS** heuristic for variable order.

Another such heuristic is called *Lexicographic search for perfect orders* [Koster et al., 2001]. Each vertex of G_φ is assigned an initially-empty set of vertices (called the *label*). At each step of the heuristic, the next variable chosen is the variable x whose label is lexicographically smallest among the unchosen variables (breaking ties arbitrarily). Then x is added to the label of its neighbors in G_φ. We call this the **LexP** heuristic for variable order.

A similar heuristic is called *Lexicographic search for minimal orders* [Koster et al., 2001]. As before, each vertex of G_φ is assigned an initially-empty label. At each step of the heuristic, the next variable chosen is again the variable x whose label is lexicographically smallest (breaking ties arbitrarily). In this case, x is added to the label of every variable y where there is a path $x, z_1, z_2, \ldots, z_k, y$ in G_φ such that every z_i is unchosen and the label of z_i is lexicographically smaller than the label of y. We call this the **LexM** heuristic for variable order.

Additionally, the variable orders produced by each of the heuristics **MCS**, **LexP**, and **LexM** can be inverted. We call these new heuristics **InvMCS**, **InvLexP**, and **InvLexM**.

4.3 Heuristics for `get-clause-rank`

The heuristic chosen for `get-clause-rank` indicates the strategy for clustering the clauses of φ. In this work, we consider three possible heuristics to be chosen for `get-clause-rank` that satisfy the conditions of Theorem 3: **Mono**, **BE**, and **BM**.

One simple case is when `get-clause-rank` is constant on all clauses, e.g., when `get-clause-rank`$(\gamma, \rho) = m$ for all $\gamma \in \varphi$. In this case, all clauses of φ are placed in Γ_m, so Algorithm 1 combines all clauses of φ into a single ADD before performing projections. This corresponds to the monolithic approach we mentioned earlier. We thus call this the **Mono** heuristic for `get-clause-rank`. Notice that the performance of Algorithm 1 with **Mono** does not depend on the heuristic for `get-cluster-var-order` or `choose-cluster`. This heuristic has previously been applied to ADDs in the context of knowledge compilation [Fargier et al., 2014].

A more complex heuristic assigns the rank of each clause to be the smallest ρ-rank of the variables that appear in the clause. That is, `get-clause-rank`$(\gamma, \rho) = \min_{x \in \texttt{Vars}(\gamma)} \rho(x)$. This heuristic corresponds to *bucket elimination* [Dechter, 1999], so we call this the **BE** heuristic. In this case, notice that every clause containing $x \in X$ can only appear in Γ_i with $i \leq \rho(x)$. It follows that x has always been projected from

all clauses by the end of iteration $\rho(x)$ in the second loop of Algorithm 1 using **BE**.

A closely related heuristic assigns the rank of each clause to be the largest ρ-rank of the variables that appear in the clause. That is, `get-clause-rank`$(\gamma, \rho) = \max_{x \in \texttt{Vars}(\gamma)} \rho(x)$. This heuristic corresponds to *Bouquet's Method* [Bouquet, 1999], so we call this the **BM** heuristic. Unlike the **BE** case, we can make no guarantee about when each variable is projected in Algorithm 1 using **BM**.

4.4 Heuristics for `choose-cluster`

The heuristic chosen for `choose-cluster` indicates the strategy for combining the ADDs produced from each cluster. In this work, we consider two possible heuristics to use for `choose-cluster` that satisfy the conditions of Theorem 3: **List** and **Tree**.

One heuristic is when `choose-cluster` selects to place A_i in the closest cluster that satisfies the conditions of Theorem 3, namely the next cluster to be processed. That is, `choose-cluster`$(A_i, i) = i+1$. Under this heuristic, Algorithm 1 equivalently builds an ADD for each cluster and then combines the ADDs in a one-by-one, in-order fashion, projecting variables as early as possible. In every iteration, there is a single intermediate ADD representing the combination of previous clusters. We call this the **List** heuristic.

Another heuristic is when `choose-cluster` selects to place A_i in the furthest cluster that satisfies the conditions of Theorem 3. That is, `choose-cluster`(A_i, i) returns the smallest $j > i$ such that $X_j \cap \texttt{Vars}(A_i) \neq \varnothing$ (or returns m, if $\texttt{Vars}(A_i) =$

$\varnothing$). In every iteration, there may be multiple intermediate ADDs representing the combinations of previous clusters. We call this the **Tree** heuristic.

Notice that the computational structure of Algorithm 1 can be represented by a tree of clusters, where cluster κ_i is a child of cluster κ_j whenever the ADD produced from κ_i is placed in κ_j (lines 14-15). These trees are always left-deep under the **List** heuristic, but they can be more complex under the **Tree** heuristic.

A *clustering heuristic* is a valid combination of a `get-clause-rank` heuristic and a `choose-cluster` heuristic. The five clustering heuristics are: **Mono**, **BE − List**, **BE − Tree**, **BM − List**, and **BM − Tree**.

4.5 Similar Techniques in Model Counting and Related Problems

Algorithm 1 performs weighted model counting with some techniques borrowed from probabilistic inference and constraint satisfaction. Although these techniques are well-known, their application to ADD-based model counting as presented in this thesis is novel.

4.5.1 Probabilistic Inference

We employ dynamic programming by caching intermediate results to avoid recomputation. This technique has previously been used in symbolic probabilistic inference in belief networks [Shachter et al., 1990].

Algorithm 1 performs model counting with early projection: iteratively eliminating

variables which do not appear in future computations. Variable elimination, a related technique, has been applied earlier to compute a posterior probability given a Bayesian network: summing out a variable from a potential at each step [Zhang and Poole, 1994]. Additional inference tasks (such as updating beliefs and finding most probable explanations) can be performed with generalized variable elimination [Dechter, 1999] using a framework similar to nonserial dynamic programming [Bertele and Brioschi, 1973]. Notice that finding an optimal order for variables to be eliminated is NP-complete [Hojati et al., 1996]. On another related note, the recursive conditioning algorithm for Bayesian inference allows time-space tradeoffs, providing more flexibility than traditional algorithms based on variable elimination and clustering [Allen and Darwiche, 2002].

In addition, both model counting and Bayesian inference can apply factorization techniques to keep intermediate computations small. In particular, Algorithm 1 takes advantage of the factored representation of a CNF formula φ as a product of the clauses in φ. Similarly, in Bayesian inference, the joint potential of a set S of potentials can be computed by multiplying all potentials in S [Zhang and Poole, 1994].

4.5.2 Constraint Satisfaction

To solve the model counting problem, we adopt some techniques from the constraint satisfaction problem (CSP) as well. For example, the Gaifman graph of a Boolean formula corresponds to the primal constraint graph of a high-order CSP instance

[Dechter and Pearl, 1989].

Also, in Algorithm 1, the function `choose-cluster` effectively creates a tree of clusters (sets of clauses) to guide the recombination of subproblems in our dynamic programming framework. In CSP, a counterpart is the tree-clustering procedure for constraint networks [Dechter and Pearl, 1989].

Chapter 5

Empirical Evaluation

We implement Algorithm 1 using the ADD package `CUDD` [Somenzi, 2015] to produce `ADDMC`, a new weighted model counter. `ADDMC` supports all heuristics described in Chapter 4. The `ADDMC` source code can be found in a public repository (`https://github.com/vardigroup/ADDMC`).

We aim to:

- find good heuristic configurations for our tool `ADDMC`, and
- compare `ADDMC` against four state-of-the-art exact model counters: `Cachet` [Sang et al., 2004], `c2d` [Darwiche, 2004], `d4` [Lagniez and Marquis, 2017], and `miniC2D` [Oztok and Darwiche, 2015].

To accomplish this, we evaluate on a set of 1914 literal-weighted CNF model counting benchmarks. These benchmarks were gathered from two sources: 1091 benchmarks with literal weights given in the interval $[0, 1]$ (`https://www.cs.rochester.edu/u/kautz/Cachet/Model_Counting_Benchmarks/`) [Sang et al., 2005], and 823 benchmarks that are originally unweighted (`http://www.cril.univ-artois.fr/KC/benchmarks.html`) [Clarke et al., 2001, Sinz et al., 2003, Palacios and Geffner, 2009, Klebanov et al., 2013]. As we focus in this work on weighted model counting, we

generate weights for these unweighted benchmarks by, for each variable x, randomly assigning either weights $W^+(x) = 0.5$ and $W^-(x) = 1.5$, or $W^+(x) = 1.5$ and $W^-(x) = 0.5$. Generating weights in this particular way results in a reasonably low amount of floating-point underflow and overflow for all model counters. (For each variable x in a formula φ, `Cachet` requires $W^+(x) + W^-(x) = 1$ unless $W^+(x) = W^-(x) = 1$. So we use weights 0.25 and 0.75 for `Cachet` and multiply the model count by $2^{|\texttt{Vars}(\varphi)|}$ as a postprocessing step.)

5.1 Experiment 1A: Comparing All `ADDMC` Heuristic Configurations

`ADDMC` heuristic configurations are constructed from five clustering heuristics (**Mono**, **BE-List**, **BE-Tree**, **BM-List**, and **BM-Tree**) together with seven variable order heuristics (**Random**, **MCS**, **InvMCS**, **LexP**, **InvLexP**, **LexM**, and **InvLexM**). Using one variable order heuristic for the cluster variable order and another for the diagram variable order gives us 245 configurations in total. We compare these configurations to find the best combination of heuristics.

On a Linux cluster with Xeon E5-2650v2 CPUs (2.60-GHz), we run each combination of heuristics on each benchmark using 24 GB of memory and a 10-second timeout. Due to the large number of `ADDMC` heuristic configurations (245), it is difficult to use a longer timeout.

5.1.1 Correctness Analysis

To compare model counts produced by different heuristic configurations of `ADDMC` (in the presence of imprecision from floating-point arithmetic), we consider non-negative real numbers $a \leq b$ equal when:

$$\begin{cases} b - a \leq 10^{-3} & \text{if } a = 0 \text{ or } b \leq 1 \\ b/a \leq 1 + 10^{-3} & \text{otherwise} \end{cases} \tag{3}$$

Even with the equality tolerance in Equation (3), different `ADDMC` heuristic configurations still produce different answers on two benchmarks, due to 64-bit floating-point imprecision.

5.1.2 Performance Analysis

Table 1 reports the best (first through fifth), median, and worst combinations across all 245 `ADDMC` heuristic configurations. See Figure 2 for a more detailed analysis of the runtime of some of these configurations. Evidently, a number of configurations perform quite well while others perform poorly. The wide range of performance indicates that the choice of heuristics is essential to the competitiveness of `ADDMC`. We also observe that Bouquet's Method and bucket elimination have similar-performing best configurations (*Best1* and *Best2*). This shows that Bouquet's Method is competitive with bucket elimination.

Best1 (**BM-Tree** clustering with **LexP** cluster variable order and **MCS** diagram variable order) is the heuristic configuration able to solve the most benchmarks in 10

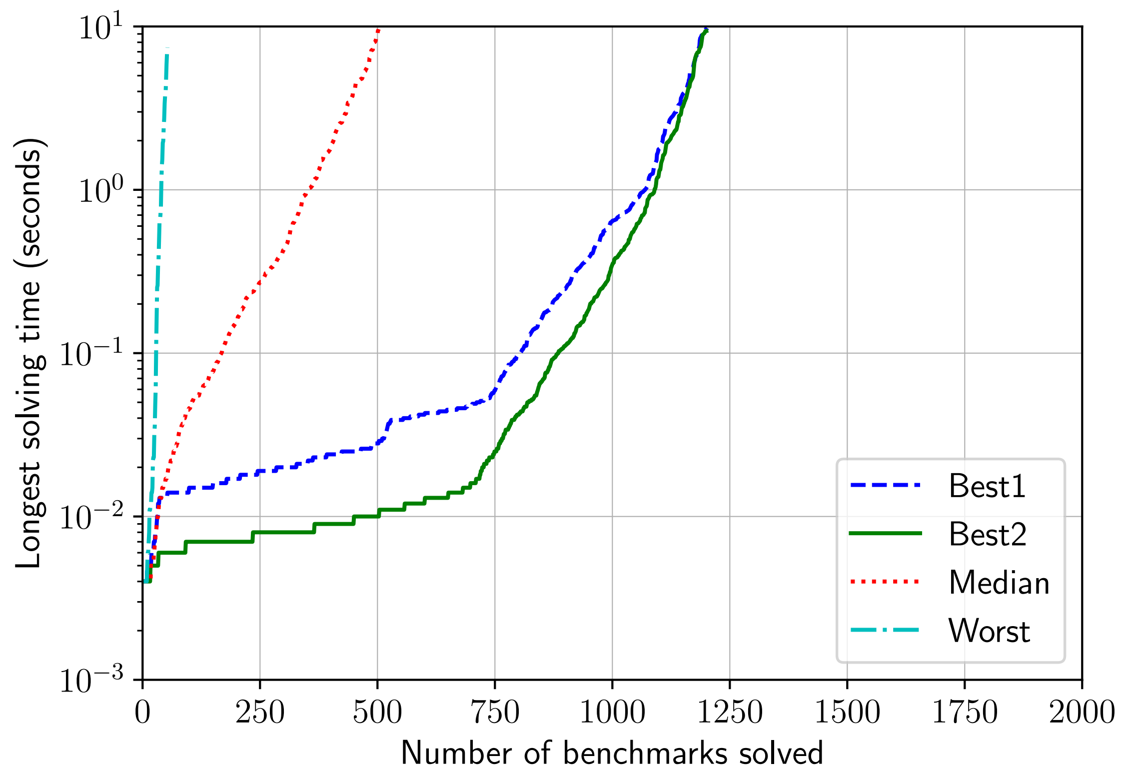


Figure 2 : Experiment 1A: a cactus plot of the numbers of benchmarks solved by the best, second-best, median, and worst `ADDMC` heuristic configurations.

Table 1 : Experiment 1A: the numbers of benchmarks solved (of 1914) in 10 seconds by the best (1st-5th), median, and worst `ADDMC` heuristic configurations.

Clustering	Cluster variable order	Diagram variable order	Solved	Name
BM-Tree	**LexP**	**MCS**	1202	*Best1*
BE-Tree	**InvLexP**	**MCS**	1200	*Best2*
BM-List	**LexP**	**MCS**	1200	*Best3*
BM-Tree	**LexP**	**InvMCS**	1199	*Best4*
BM-List	**LexP**	**InvMCS**	1197	*Best5*
BE-List	**LexP**	**LexP**	504	*Median*
BE-List	**Random**	**Random**	53	*Worst*

seconds. To see how the five best configurations here perform in a longer timeout, we conduct Experiment 1B.

5.2 Experiment 1B: Comparing Best `ADDMC` Heuristic Configurations

Previously, Experiment 1A compares 245 `ADDMC` heuristic configurations in 10 seconds (Table 1). Now, Experiment 1B selects the five best configurations in Experiment 1A and compares them in 1000 seconds. On a Linux cluster with Xeon E5-2650v2 CPUs (2.60-GHz), we run each of these five configurations on each benchmark using 24 GB of memory.

5.2.1 Correctness Analysis

Even with the aforementioned floating-point equality tolerance in Equation (3), different `ADDMC` heuristic configurations still produce different answers on seven benchmarks (of 1914), due to 64-bit floating-point imprecision.

5.2.2 Performance Analysis

Previously, Table 1 shows the performance of the five best `ADDMC` heuristic configurations in 10 seconds (245 configurations are evaluated in total). Now, Table 2 compares those five configurations in 1000 seconds as well as their *virtual best solver (VBS5)*. For each benchmark, the runtime of *VBS5* is the shortest runtime across the five actual configurations.

Interestingly, there are several inversions between Table 1 and Table 2. For instance, *Best1* (the configuration that solves the most benchmarks in Experiment 1A) only solves the fourth-most benchmarks in Experiment 1B. If we compare all 245 configurations in an even longer timeout (e.g., 10000-second), then the result may change again. Therefore, we decide use *Best1* as the default configuration for `ADDMC` in Experiment 2, as Experiment 1A compares all configurations whereas Experiment 1B only compares five.

See Figure 3 for a more detailed analysis of the runtime of these five configurations and *VBS5*. We see that *Best2* performs similarly to *VBS5* on the first 1000 benchmarks. But after that, *VBS5* is significantly better than all actual configurations.

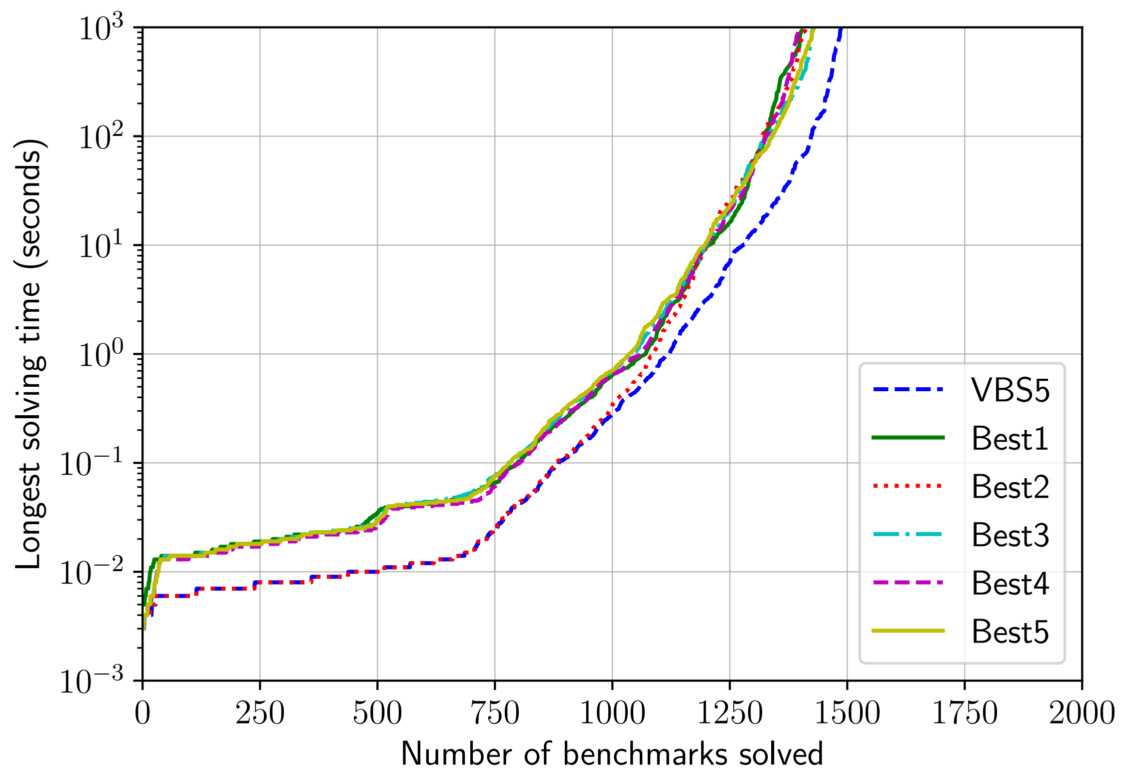


Figure 3 : Experiment 1B: a cactus plot of the numbers of benchmarks solved in 1000 seconds by the five best ADDMC heuristic configurations in Experiment 1A (10 seconds) and their virtual best solver (*VBS5*).

Table 2 : Experiment 1B: the numbers of benchmarks solved (of 1914) in 1000 seconds by the five best `ADDMC` heuristic configurations in Experiment 1A (10 seconds) and their virtual best solver (*VBS5*).

Clustering	Cluster variable order	Diagram variable order	Solved	Name
–	–	–	1489	*VBS5*
BM-List	**LexP**	**InvMCS**	1430	*Best5*
BM-List	**LexP**	**MCS**	1427	*Best3*
BE-Tree	**InvLexP**	**MCS**	1413	*Best2*
BM-Tree	**LexP**	**MCS**	1403	*Best1*
BM-Tree	**LexP**	**InvMCS**	1396	*Best4*

5.3 Experiment 2: Comparing `ADDMC` to Weighted Model Counters

In Experiment 1A, the `ADDMC` heuristic configuration able to solve the most benchmarks is *Best1* (**BM-Tree** clustering with **LexP** cluster variable order and **MCS** diagram variable order). Using this as the default configuration, we now compare `ADDMC` to four state-of-the-art weighted model counters: `Cachet`, `c2d`, `d4`, and `miniC2D`. We note that `Cachet` uses `long double` precision, whereas all other model counters use `double` precision. Also, `c2d` does not natively support weighted model counting. In order to compare `c2d` to weighted model counters, we constructed a simple weighted model counter that uses `c2d` to compile CNF into d-DNNF and then uses `d-DNNF-reasoner` (`http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html`) to compute the weighted model count. On average, `c2d`'s compilation time is 81.65% of the total

time.

On a Linux cluster with Xeon E5-2650v2 CPUs (2.60-GHz), we run each counter on each benchmark using 24 GB of memory and a 1000-second timeout.

5.3.1 Correctness Analysis

Even with the aforementioned floating-point equality tolerance in Equation (3), weighted model counters still sometimes produce different answers for the same benchmark due to floating-point effects. In particular, of 1008 benchmarks that are solved by all five model counters, `ADDMC` produces 7 model counts that differ from the output of all four other tools. For `Cachet`, `c2d`, `d4`, and `miniC2D`, the numbers are respectively 55, 0, 42, and 0. To improve `ADDMC`'s precision, we plan as future work to integrate a new decision diagram package, `Sylvan` [van Dijk and van de Pol, 2015], into `ADDMC`. `Sylvan` can interface with the GNU Multiple Precision library to support ADDs with higher-precision numbers.

5.3.2 Performance Analysis

As described earlier, we use 1914 benchmarks to compare five weighted model counters (`ADDMC`, `Cachet`, `c2d`, `d4`, and `miniC2D`). We also report the performance of the virtual best solver (`VBS1`). For each benchmark, the runtime of `VBS1` is the shortest runtime across all five actual solvers. To quantify how much `ADDMC` improves the portfolio of actual solvers, we additionally introduce `VBS0`, whose runtime for each benchmark is the shortest runtime across four existing tools (`Cachet`, `c2d`, `d4`, and `miniC2D`)

Table 3 : Experiment 2: the numbers of benchmarks solved (of 1914) in 1000 seconds — uniquely (i.e., benchmarks solved by no other solver), fastest, and in total — by five weighted model counters and two virtual best solvers (`VBS1` and `VBS0`).

Solvers	Benchmarks solved		
	Unique	Fastest	Total
`VBS1` (with `ADDMC`)	–	–	1771
`VBS0` (without `ADDMC`)	–	–	1647
`d4`	12	283	1587
`c2d`	0	13	1417
`miniC2D`	8	61	1407
`ADDMC`	124	763	1404
`Cachet`	14	651	1383

without `ADDMC`. Table 3 summarizes the performance of five weighted model counters, `VBS1`, and `VBS0`. `ADDMC` is fastest on 763 benchmarks, including 124 benchmarks solved by no other tool.

See Figure 4 for a detailed analysis of the solvers' runtime. Although solving fewer benchmarks overall than `d4`, our tool `ADDMC` improves the VBS on 763 benchmarks. Moreover, `ADDMC` is able to solve 124 benchmarks that no other weighted model counter we consider can solve. We conclude that `ADDMC` is a useful addition to the portfolio of weighted model counters.

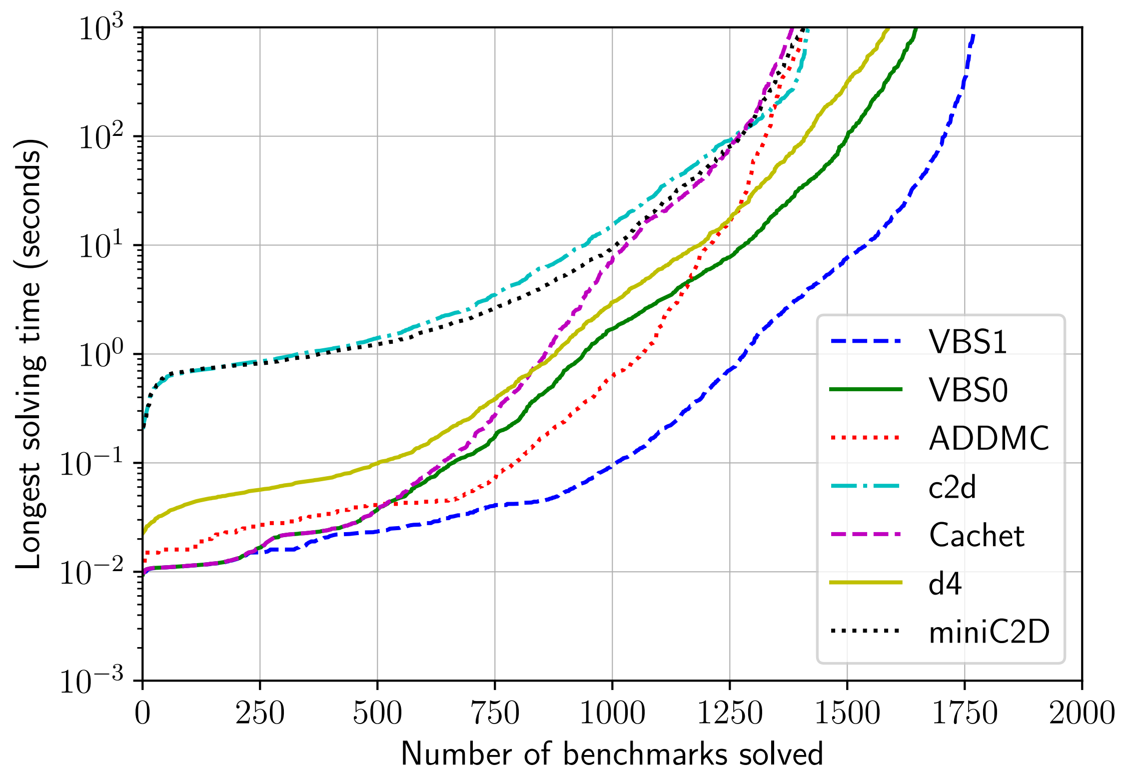


Figure 4 : Experiment 2: a cactus plot of the numbers of benchmarks solved by five weighted model counters and two virtual best solvers (`VBS1` with `ADDMC` and `VBS0` without `ADDMC`).

5.3.3 Predicting ADDMC Performance

Generally, ADDMC can solve a benchmark quickly if all intermediate ADDs constructed during the model counting process are small. An ADD is small when it achieves high compression under a good diagram variable order; predicting this a priori is difficult and is an area of active research. However, an ADD also tends to be small if it has few variables, which occurs when an ADDMC heuristic configuration results in many opportunities for early projection. Moreover, the number of variables that occur in each ADD produced by Algorithm 1 can be computed much faster than computing the full model count (since we do not need to actually construct the ADDs).

Formally, fix an ADDMC heuristic configuration. For a given benchmark, define the *maximum ADD variable count (MAVC)* to be the largest number of variables across all ADDs that would be constructed when running Algorithm 1. Using the heuristic configuration of Experiment 2 (*Best1*), we were able to compute the MAVCs of 1906 benchmarks (of 1914 in total). We were unable to compute the MAVCs of the remaining 8 benchmarks within 10000 seconds due to the large number of variables and clauses; these benchmarks were also not solved by ADDMC.

Figure 5 shows the number of benchmarks solved by ADDMC in Experiment 2 for various upper bounds on the MAVC. Generally, ADDMC performed well on benchmarks with low MAVCs. In particular, ADDMC solved most benchmarks (1345 of 1425) with MAVCs less than 70 but solved solved few benchmarks (12 of 379) with MAVCs greater than 100.

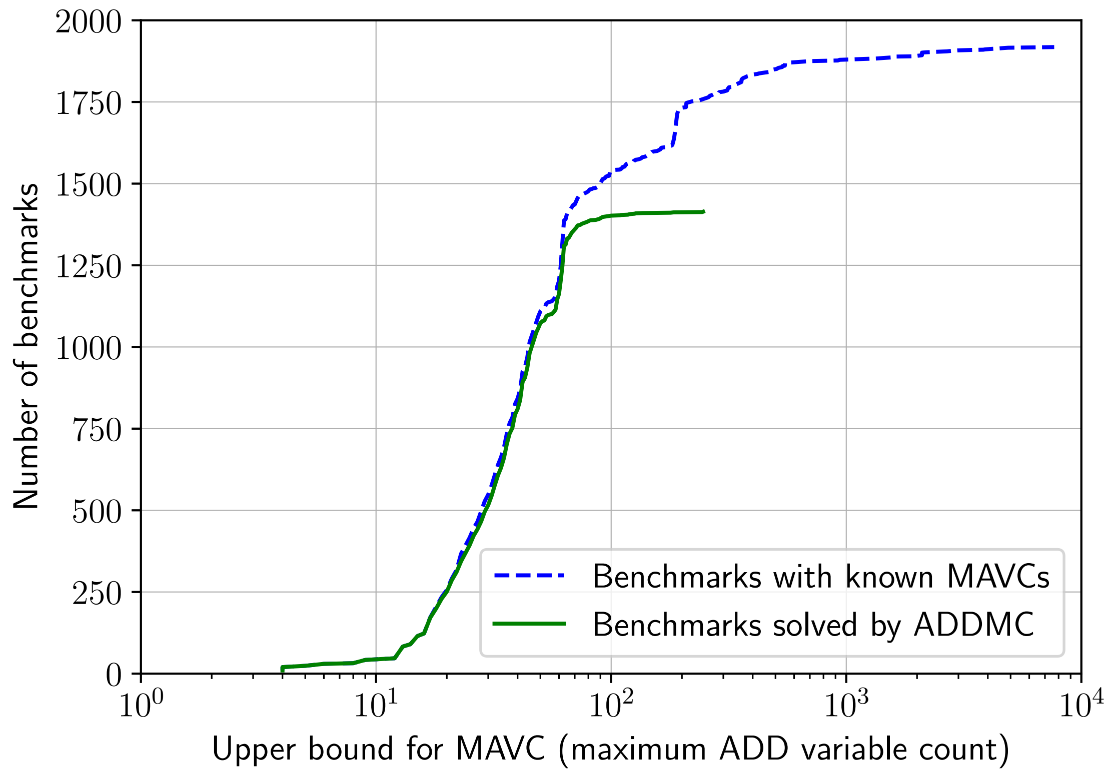


Figure 5 : A cactus plot of the number of benchmarks, in total and solved by `ADDMC`, for various upper bounds for MAVCs. The MAVCs of the 1404 benchmarks solved by `ADDMC` within 1000 seconds range from 4 to 246.

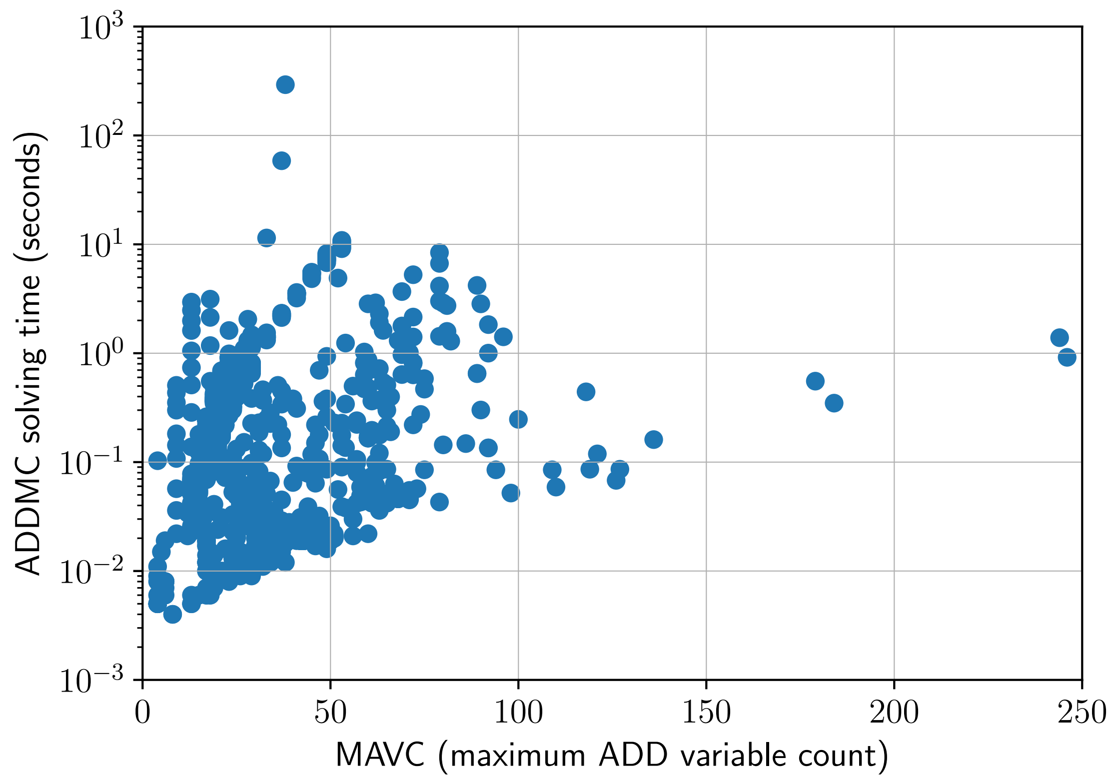


Figure 6 : A scatter plot of the solving time of `ADDMC` against the MAVC for each of the 1404 benchmarks solved by `ADDMC` within 1000 seconds.

Figure 6 shows the runtime of ADDMC on the 1404 benchmarks ADDMC was able to solve in Experiment 2. In general, ADDMC was slower on benchmarks with higher MAVCs.

From these two observations, we conclude that the MAVC of a benchmark (under a particular heuristic configuration) is a good predictor of ADDMC performance.

5.4 Experiment 3: Comparing ADDMC to Bayesian Inference Engines

Our experiments use 1914 model counting benchmarks from two classes. In particular, the **Bayes** benchmark class contains 1091 Bayesian inference instances encoded as CNF literal-weighted model counting instances [Sang et al., 2005].

For future work, using these **Bayes** benchmarks, we plan to compare two approaches to Bayesian inference:

1. using specialized Bayesian inference engines

2. reducing Bayesian inference to model counting then using model counters

For the first approach, there are some dedicated Bayesian network solvers:

- JavaBayes (https://www.cs.cmu.edu/~javabayes/)

- Netica (https://www.norsys.com/netica.html)

- SamIam (http://reasoning.cs.ucla.edu/samiam/)

- `Valelim` (no longer available)

For the second approach, we can first convert Bayesian networks to CNF formulas using the tool `bif2cnf` (`https://www.cs.rochester.edu/u/kautz/Cachet/Model_Counting_Benchmarks/`). Then we can use `ADDMC` to compute weighted model counts of the formulas.

Performing Bayesian inference via reduction to weighted model counting can be faster than directly solving Bayesian networks. In particular, the model counter `Cachet` outperforms dedicated Bayesian solvers `Netica`, `SamIam`, and `Valelim` in certain cases [Sang et al., 2005]. As our model counter `ADDMC` has better performance than `Cachet` in Experiment 2, we expect `ADDMC` to be competitive with Bayesian inference engines as well.

Chapter 6

Discussion

We developed a dynamic-programming framework for weighted model counting that captures both bucket elimination and Bouquet's Method. We implemented this algorithm in ADDMC, a new weighted model counter. We used ADDMC to compare bucket elimination and Bouquet's Method across a variety of variable order heuristics on 1914 standard model counting benchmarks and concluded that Bouquet's Method is competitive with bucket elimination.

Moreover, we demonstrated that ADDMC is competitive with existing state-of-the-art weighted model counters on these 1914 benchmarks. In particular, adding ADDMC allows the virtual best solver to solve 124 more benchmarks. Thus ADDMC is valuable as part of a portfolio of solvers, and ADD-based approaches to model counting in general are promising and deserve further study. One direction for future work is to investigate how benchmark properties (e.g., treewidth) correlate with the performance of ADD-based approaches to model counting. Predicting the performance of tools on CNF benchmarks is an active area of research in the SAT solving community [Xu et al., 2008].

Bucket elimination has been well-studied theoretically, with close connections to treewidth and tree decompositions. For instance, it is widely known that bucket

elimination's time and space consumption is exponential in the treewidth of a Bayesian network [Dechter, 1999, Chavira and Darwiche, 2007]. On the other hand, Bouquet's Method is much less well-known. One of the few related studies observes that Bouquet's Method can outperform bucket elimination [Pan and Vardi, 2004]. Another direction for our future work is to develop a theoretical framework to explain the relative performance between bucket elimination and Bouquet's Method.

In this work, we focused on ADDs implemented in the ADD package `CUDD` [Somenzi, 2015]. There are other ADD packages that may be fruitful to explore in the future. For example, `Sylvan` supports multicore operations on ADDs, which would allow us to investigate the impact of parallelism on our techniques. Moreover, `Sylvan` supports arbitrary-precision arithmetic [van Dijk and van de Pol, 2015].

Other compact representations have been used in dynamic-programming frameworks for related problems. For example, AND/OR multi-valued decision diagrams [Mateescu et al., 2008], probabilistic sentential decision diagrams [Shen et al., 2016], and probabilistic decision graphs [Jaeger, 2004] have all been used for Bayesian inference. Moreover, weighted decision diagrams have been used for optimization [Hooker, 2013], and affine ADDs have been used for planning [Sanner and McAllester, 2005]. It would be interesting to see if these compact representations also improve dynamic-programming frameworks for model counting.

Another future research direction is to explore new ways to build and combine clusters in Algorithm 1. A promising technique is to use tree decompositions, which

have been shown to work for the #P-hard problem of tensor-network contraction [Dudek et al., 2019].

Bibliography

[Aguirre and Vardi, 2001] Aguirre, A. S. M. and Vardi, M. Y. (2001). Random 3-SAT and BDDs: the plot thickens further. In *CP*, pages 121–136.

[Allen and Darwiche, 2002] Allen, D. and Darwiche, A. (2002). New advances in inference by recursive conditioning. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*, pages 2–10. Morgan Kaufmann Publishers Inc.

[Bacchus et al., 2009] Bacchus, F., Dalmao, S., and Pitassi, T. (2009). Solving #SAT and Bayesian inference with backtracking search. *JAIR*, 34:391–442.

[Bahar et al., 1997] Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Form Method Syst Des*, 10(2-3):171–206.

[Bertele and Brioschi, 1973] Bertele, U. and Brioschi, F. (1973). On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148.

[Biere et al., 2009] Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS Press.

[Bouquet, 1999] Bouquet, F. (1999). *Gestion de la dynamicité et énumération d'impliquants premiers: une approche fondée sur les diagrammes de décision binaire*. PhD thesis, Aix-Marseille 1.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE TC*, 35(8).

[Burch et al., 1991] Burch, J., Clarke, E., and Long, D. (1991). Symbolic model checking with partitioned transition relations. In *VLSI*, pages 49–58.

[Chavira and Darwiche, 2007] Chavira, M. and Darwiche, A. (2007). Compiling Bayesian networks using variable elimination. In *IJCAI*, pages 2443–2449.

[Clarke et al., 2001] Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Form Method Syst Des*, 19(1):7–34.

[Darwiche, 2004] Darwiche, A. (2004). New advances in compiling CNF into decomposable negation normal form. In *ECAI*, pages 328–332.

[Darwiche, 2011] Darwiche, A. (2011). SDD: a new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826.

[Dechter, 1999] Dechter, R. (1999). Bucket elimination: a unifying framework for reasoning. *AI*, 113(1-2):41–85.

[Dechter and Pearl, 1989] Dechter, R. and Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366.

[Domshlak and Hoffmann, 2007] Domshlak, C. and Hoffmann, J. (2007). Probabilistic planning via heuristic forward search and weighted model counting. *JAIR*, 30:565–620.

[Dudek et al., 2019] Dudek, J. M., Dueñas-Osorio, L., and Vardi, M. Y. (2019). Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *arXiv preprint arXiv:1908.04381*.

[Dudek et al., 2020] Dudek, J. M., Phan, V. H., and Vardi, M. Y. (2020). ADDMC: weighted model counting with algebraic decision diagrams. In *AAAI*.

[Fargier et al., 2014] Fargier, H., Marquis, P., Niveau, A., and Schmidt, N. (2014). A knowledge compilation map for ordered real-valued decision diagrams. In *AAAI*, pages 1049–1055.

[Gogate and Domingos, 2012] Gogate, V. and Domingos, P. (2012). Approximation by quantization. *arXiv:1202.3723*.

[Hoey et al., 1999] Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD: stochastic planning using decision diagrams. In *UAI*, pages 279–288.

[Hojati et al., 1996] Hojati, R., Krishnan, S. C., and Brayton, R. K. (1996). Early quantification and partitioned transition relations. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 12–19. IEEE.

[Hooker, 2013] Hooker, J. N. (2013). Decision diagrams and dynamic programming. In *CPAIOR*, pages 94–110.

[Howard, 1966] Howard, R. A. (1966). Dynamic programming. *JMS*.

[Jaeger, 2004] Jaeger, M. (2004). Probabilistic decision graphs – combining verification and AI techniques for probabilistic inference. *IJUFKS*, 12(supp01):19–42.

[Klebanov et al., 2013] Klebanov, V., Manthey, N., and Muise, C. (2013). SAT-based analysis and quantification of information flow in programs. In *QEST*, pages 177–192.

[Kolaitis and Vardi, 2000] Kolaitis, P. G. and Vardi, M. Y. (2000). Conjunctive-query containment and constraint satisfaction. *JCSS*, 61(2):302–332.

[Koster et al., 2001] Koster, A. M., Bodlaender, H. L., and Van Hoesel, S. P. (2001). Treewidth: computational experiments. *Electron Notes Discrete Math*, 8:54–57.

[Kwiatkowska et al., 2007] Kwiatkowska, M., Norman, G., and Parker, D. (2007). Stochastic model checking. In *SFM*, pages 220–270.

[Lagniez and Marquis, 2017] Lagniez, J.-M. and Marquis, P. (2017). An improved decision-DNNF compiler. In *IJCAI*, pages 667–673.

[Mateescu et al., 2008] Mateescu, R., Dechter, R., and Marinescu, R. (2008). AND/OR multi-valued decision diagrams for graphical models. *JAIR*, 33:465–519.

[McMahan et al., 2004] McMahan, B. J., Pan, G., Porter, P., and Vardi, M. Y. (2004). Projection pushing revisited. In *EDBT*, pages 441–458.

[Naveh et al., 2007] Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcu, E., and Shurek, G. (2007). Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 28(3):13–13.

[Oztok and Darwiche, 2015] Oztok, U. and Darwiche, A. (2015). A top-down compiler for sentential decision diagrams. In *IJCAI*, pages 3141–3148.

[Palacios and Geffner, 2009] Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *JAIR*, 35:623–675.

[Pan and Vardi, 2005] Pan, G. and Vardi, M. (2005). Symbolic techniques in satisfiability solving. *J Autom Reason*, 35(1-3):25–50.

[Pan and Vardi, 2004] Pan, G. and Vardi, M. Y. (2004). Search vs. symbolic techniques in satisfiability solving. In *SAT*, pages 235–250.

[Poole and Zhang, 2003] Poole, D. and Zhang, N. L. (2003). Exploiting contextual independence in probabilistic inference. *JAIR*, 18:263–313.

[Samer and Szeider, 2010] Samer, M. and Szeider, S. (2010). Algorithms for propositional model counting. *J Discrete Algorithms*, 8(1):50–64.

[Sang et al., 2004] Sang, T., Bacchus, F., Beame, P., Kautz, H. A., and Pitassi, T. (2004). Combining component caching and clause learning for effective model counting. *SAT*, pages 20–28.

[Sang et al., 2005] Sang, T., Beame, P., and Kautz, H. (2005). Solving Bayesian networks by weighted model counting. In *AAAI*, volume 1, pages 475–482. AAAI Press.

[Sanner and McAllester, 2005] Sanner, S. and McAllester, D. (2005). Affine algebraic decision diagrams and their application to structured probabilistic inference. In *IJCAI*, pages 1384–1390.

[Shachter et al., 1990] Shachter, R. D., D'Ambrosio, B., and Del Favero, B. (1990). Symbolic probabilistic inference in belief networks. In *AAAI*, volume 90, pages 126–131.

[Shen et al., 2016] Shen, Y., Choi, A., and Darwiche, A. (2016). Tractable operations for arithmetic circuits of probabilistic models. In *Adv Neural Inf Process Syst*, pages 3936–3944.

[Shenoy and Shafer, 2008] Shenoy, P. P. and Shafer, G. (2008). Axioms for probability and belief-function propagation. In *Classic Works of the Dempster-Shafer Theory of Belief Functions*, pages 499–528. Springer.

[Sinz et al., 2003] Sinz, C., Kaiser, A., and Küchlin, W. (2003). Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97.

[Somenzi, 2015] Somenzi, F. (2015). *CUDD: CU decision diagram package - release 3.0.0*. University of Colorado at Boulder.

[Tarjan and Yannakakis, 1984] Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SICOMP*, 13(3):566–579.

[Uribe and Stickel, 1994] Uribe, T. E. and Stickel, M. E. (1994). Ordered binary decision diagrams and the Davis-Putnam procedure. In *CCL*, pages 34–49.

[Valiant, 1979] Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *SICOMP*, 8(3):410–421.

[van Dijk and van de Pol, 2015] van Dijk, T. and van de Pol, J. (2015). Sylvan: multi-core decision diagrams. In *TACAS*, pages 677–691.

[Wilson and Mengin, 1999] Wilson, N. and Mengin, J. (1999). Logical deduction using the local computation framework. In *ECSQARU*, pages 386–396.

[Xu et al., 2008] Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606.

[Zhang and Poole, 1994] Zhang, N. L. and Poole, D. (1994). A simple approach to Bayesian network computations. In *Canadian AI*, pages 171–178.