ADDMC: Weighted Model Counting with Algebraic Decision Diagrams

Jeffrey M. Dudek, Vu H. N. Phan, Moshe Y. Vardi
Rice University
6100 Main Street
Houston, Texas 77005
{jmd11, vhp1, vardi}@rice.edu

Abstract
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 main data structure. We implement this technique in 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 weighted model counters (Cachet, c2d, d4, and miniC2D) on 1914 standard model counting benchmarks and show that ADDMC significantly improves the virtual best solver.

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, Heule, and van Maaren 2009; Naveh et al. 2007). Given an input set of constraints, with the focus in this paper 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, Dalmao, and Pitassi 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, Dalmao, and Pitassi 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, Dalmao, and Pitassi 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, an 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 or-
ders may produce ADDs that are exponentially smaller than others for the same real-valued function. A variety of tech-
iques exist in prior work to heuristically find diagram vari-
able orders (Tarjan and Yannakakis 1984; Koster, Bodlaen-
der, and Van Hoesel 2001). In addition to the diagram vari-
able 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 vari-
able order. We show that the choice of heuristics to find
cluster variable orders has a significant impact on the run-
time 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 satisfiabil-
ity 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
tics 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 of 1914 benchmarks.

2 Preliminaries

In this section, we introduce weighted model counting, the
central problem of this work, and Algebraic Decision Dia-
grams, 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 \( \varphi : 2^X \rightarrow \{0, 1\} \) be a Boolean function
over a set \( X \) of variables, and let \( W : 2^X \rightarrow \mathbb{R} \) be an ar-
bitrary function. The \textit{weighted model count} of \( \varphi \) w.r.t. \( W \)
is

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

The function \( W : 2^X \rightarrow \mathbb{R} \) is called a \textit{weight func-
tion}. In this work, we focus on so-called \textit{literal-weight func-
tions}, 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 \( W_+(x), W_-(x) : X \rightarrow \mathbb{R} \). One can in-
terpret these literal-weight functions \( W \) as assigning a real-
valued weight to each literal: \( W_+(x) \) to \( x \) and \( W_-(x) \) to
\( \lnot x \). It is common to restrict attention to weight functions
whose range is \( \mathbb{R} \) or just the interval \([0, 1]\).

When the formula \( \varphi \) is given in \textit{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 re-
presentation. For example, Cachet uses DPLL search com-
bined with component caching and clause learning to per-
form weighted model counting (Sang et al. 2004).

If \( \varphi \) is given in a compact representation — e.g., as a Bi-
ary Decision Diagram (BDD) (Bryant 1986) or as a Sen-
tential Decision Diagram (SDD) (Darwiche 2011) — com-
puting 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). Although usually more succinct than a
truth table, such compact representations may still be ex-
ponential in the size of the CNF formula in the worst case
(Bova et al. 2016).

2.2 Algebraic Decision Diagrams

The central data structure we use in this work is \textit{Algebraic
Decision Diagram} (ADD) (Bahar et al. 1997), a compact
representation of a function as a directed acyclic graph. For-
manly, an ADD is a tuple \((X, S, \pi, G)\), where \( X \) is a set of
Boolean variables, \( S \) is an arbitrary set (called the \textit{carrier
set}), \( \pi : X \rightarrow \mathbb{Z}^+ \) is an injection (called the \textit{diagram vari-
able order}), and \( G \) is a rooted directed acyclic graph sat-is-
fying 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 \( \pi \). ADDs were originally designed
for matrix multiplication and shortest path algorithms (Ba-
har et al. 1997). ADDs have also been used for stochastic
model checking (Kwiatkowska, Norman, and Parker 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 \((X, S, \pi, G)\) is a compact representation of a func-
tion \( f : 2^X \rightarrow S \). Although there are many ADDs rep-resenting each such function \( f \), for each injection \( \pi : X \rightarrow
\mathbb{Z}^+ \), there is a unique minimal ADD that represents \( f \) with
\( \pi \) as the diagram variable order, called the \textit{canonical ADD}.
ADDs can be minimized in polynomial time, so it is typi-
cal to only work with canonical ADDs. Given two ADDs rep-
resenting 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 ex-
ist 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 signif-
ificant overlap with the techniques to find variable orders for
BDDs.
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.

3 Using ADDs for Weighted Model Counting with Early Projection

An ADD with carrier set $\mathbb{R}$ can be used to represent both a Boolean formula $\varphi : 2^X \to \{0, 1\}$ 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 section, 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 $\varphi$ and weight functions $W$.

3.1 General Weighted Model Counting

We assume that the Boolean formula $\varphi$ and the weight function $W$ are represented as ADDs. The goal is to compute $W(\varphi)$, the weighted model count of $\varphi$ 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 $1 : 2^\emptyset \to \mathbb{R}$ (that maps $\emptyset$ to 1). If $\varphi : 2^X \to \{0, 1\}$ and $\psi : 2^Y \to \{0, 1\}$ are Boolean formulas, the product $\varphi \cdot \psi$ is the Boolean function corresponding to the conjunction $\varphi \land \psi$.

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” $x$. Variable projection in real-valued functions is similar to variable elimination in Bayesian networks (Zhang and Poole 1994).

**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., that $\exists_x \exists_y A = \exists_y \exists_x A$ 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 formula over a set $X$ of variables, and let $W : 2^X \to \mathbb{R}$ be an arbitrary weight function. Then

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

Theorem 1 suggests that $W(\varphi)$ can be computed by constructing an ADD for $\varphi$ 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 Early Projection

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, Clarke, and Long 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 $X' \subseteq X \setminus Y$,

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

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

Assume that $\varphi : 2^X \to \{0, 1\}$ is a CNF formula, i.e., given as a set of clauses. For every clause $\gamma \in \varphi$, observe that $\gamma$ is a Boolean formula $\gamma : 2^X \to \{0, 1\}$ where $X_\gamma \subseteq X$ is the set of variables appearing in $\gamma$. 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 $\emptyset$ to $W^{-}(x)$ and $\{x\}$ to $W^{+}(x)$. One can check using Definition 2 that $W = \prod_{x \in X} W_x$.

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

$$W(\varphi) = \left(\exists_X \prod_{\gamma \in \varphi} \prod_{x \in X} W_x\right)(\emptyset).$$

(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 section.
Algorithm 1: ADD Literal-Weighted CNF Model Counting

Input: $X$: set of Boolean variables
Input: $\phi$: nonempty CNF formula over $X$
Input: $W$: literal-weight function over $X$

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

1. $\pi \leftarrow \text{get-diagram-var-order}(\phi)$ /* injection $\pi: X \rightarrow \mathbb{Z}^+$ */
2. $\rho \leftarrow \text{get-cluster-var-order}(\phi)$ /* injection $\rho: X \rightarrow \mathbb{Z}^+$ */
3. $m \leftarrow \max_{x \in X} \rho(x)$
4. for $i = m, m - 1, \ldots, 1$
   5. $\Gamma_i \leftarrow \{ \gamma \in \phi: \text{get-clause-rank}(\gamma, \rho) = i \}$ /* collecting clauses $\gamma$ with rank $i$ */
   6. $\kappa_i \leftarrow \{ \text{get-clause-ADD}(\gamma, \pi): \gamma \in \Gamma_i \}$ /* cluster $\kappa_i$ contains ADDs of clauses $\gamma$ with rank $i$ */
   7. $X_i \leftarrow \text{Vars}(\kappa_i) \setminus \bigcup_{p=i+1}^m \text{Vars}(\kappa_p)$ /* variables already placed in $X_i$ will not be placed in $X_1, X_2, \ldots, X_{i-1}$ */
8. for $i = 1, 2, \ldots, m$
   9. if $\kappa_i \neq \emptyset$
      10. $A_i \leftarrow \prod_{D \in \kappa_i} D$
         for $x \in X_i$
         12. $A_i \leftarrow \exists_x (A_i \cdot W_x)$
         13. if $i < m$
         14. $j \leftarrow \text{choose-cluster}(A_i, i)$
         15. $\kappa_j \leftarrow \kappa_j \cup \{A_i\}$
16. return $A_m(\emptyset)$ /* product of all ADDs $D$ in cluster $\kappa_i$ */
   /* $W_x: 2^x \rightarrow \mathbb{R}$, represented by an ADD */
   /* $i < j \leq m$ */
   /* $A_m: 2^\emptyset \rightarrow \mathbb{R}$ */

4 Dynamic Programming for Model Counting

In this section, we 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 $\phi$ into clusters. For each cluster, we construct the ADD corresponding to the conjunction of its clauses. These ADDs are then incrementally combined via the multiplication operator. Throughout, each variable of the ADD is projected as early as Theorem 3. The heuristic chosen for Algorithm 1 to implement $\text{get-diagram-var-order}$, $\text{get-cluster-var-order}$, $\text{get-clause-rank}$, and $\text{choose-cluster}$ calls satisfy the following conditions:

1. $1 \leq \text{get-clause-rank}(\gamma, \rho) \leq m$,
2. $i < \text{choose-cluster}(A_i, i) \leq m$, and
3. $X_i \cap \text{Vars}(A_i) = \emptyset$ for all integers $s$ such that $i < s < \text{choose-cluster}(A_i, i)$. Then Algorithm 1 returns $W(\phi)$.

By Condition 1, we know the set $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ forms a partition of the clauses in $\phi$. 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 $\kappa_s$ 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 $\phi$ w.r.t. $W$. All heuristics we describe in this paper satisfy the conditions of Theorem 3.

In the remainder of this section, we discuss a variety of existing heuristics that can be used as a part of Algorithm 1 to implement $\text{get-diagram-var-order}$, $\text{get-cluster-var-order}$, $\text{get-clause-rank}$, and $\text{choose-cluster}$.

4.1 Heuristics for $\text{get-diagram-var-order}$ and $\text{get-cluster-var-order}$

The heuristic chosen for $\text{get-diagram-var-order}$ indicates the variable order that is used as the diagram variable order in every ADD constructed by Algorithm 1. The heuristic chosen for $\text{get-cluster-var-order}$ indicates the variable order which, when combined with the heuristic for $\text{get-clause-rank}$, is used to order the clauses of $\phi$. (BE 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.
One 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 $G_\varphi$ of a formula $\varphi$. The Gaifman graph of $\varphi$ has a vertex for every variable in $\varphi$. Two vertices are connected by an edge if and only if the corresponding variables appear in the same clause of $\varphi$.

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 $G_\varphi$ 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, Bodlaender, and Van Hoesel 2001). Each vertex of $G_\varphi$ 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 $G_\varphi$. We call this the LexP heuristic for variable order.

A similar heuristic is called Lexicographic search for minimal orders (Koster, Bodlaender, and Van Hoesel 2001). As before, each vertex of $G_\varphi$ 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 $G_\varphi$ 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.

### 4.2 Heuristics for get-clause-rank

The heuristic chosen for get-clause-rank indicates the strategy used for clustering the clauses of $\varphi$. 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 the rank of each clause is constant, e.g., when $\text{get-clause-rank}(\gamma, \rho) = m$ for all $\gamma \in \varphi$. In this case, all clauses of $\varphi$ are placed in $\Gamma_m$, so Algorithm 1 combines all clauses of $\varphi$ 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).

Another, more complex heuristic assigns the rank of each clause to be the smallest $\rho$-rank of the variables that appear in the clause. That is, $\text{get-clause-rank}(\gamma, \rho) = \min_{\vars(\gamma)} \rho(x)$. 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 $\Gamma_i$ 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 related heuristic assigns the rank of each clause to be the largest $\rho$-rank of the variables that appear in the clause. That is, $\text{get-clause-rank}(\gamma, \rho) = \max_{\vars(\gamma)} \rho(x)$. 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.

### 4.3 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, $\text{choose-cluster}(A_i, i) = i + 1$. 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, $\text{choose-cluster}(A_i, i)$ returns the smallest $j > i$ such that $X_j \cap \vars(A_i) \neq \emptyset$ (or returns $m$, if $\vars(A_i) = \emptyset$). 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 $\kappa_i$ is a child of cluster $\kappa_j$ whenever the ADD produced from $\kappa_i$ is placed in $\kappa_j$ (lines 14-15). These trees are always left-deep under the List heuristic, but they can be more complex under the Tree heuristic.

We can combine get-clause-rank heuristics and (if applicable) choose-cluster heuristics to form clustering heuristics: Mono, BE – List, BE – Tree, BM – List, and BM – Tree.

## 5 Empirical Evaluation

We implement Algorithm 1 using the ADD package CUDD to produce ADDMC, a new weighted model counter. ADDMC supports all heuristics described in Section 4. The ADDMC source code and experimental data can be obtained from a public repository (https://github.com/vardigroup/ADDMC).

We aim to: (1) find good heuristic configurations for our tool ADDMC, and (2) compare ADDMC against four state-of-the-art weighted model counters: Cachet (Sang et al. 2004), c2d (Darwiche 2004), d4 (Lagniez and Marquis...
Table 1: The numbers of benchmarks solved (of 1914) in 10 seconds by the best, second-best, median, and worst ADDMC heuristic configurations.

<table>
<thead>
<tr>
<th>Clusterings</th>
<th>Clus. var.</th>
<th>Diag. var.</th>
<th>Solved</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM-Tree</td>
<td>LexP</td>
<td>MCS</td>
<td>1202</td>
<td>Best1</td>
</tr>
<tr>
<td>BE-Tree</td>
<td>InvLexP</td>
<td>MCS</td>
<td>1200</td>
<td>Best2</td>
</tr>
<tr>
<td>BE-List</td>
<td>LexP</td>
<td>LexP</td>
<td>504</td>
<td>Median</td>
</tr>
<tr>
<td>BE-List</td>
<td>Random</td>
<td>Random</td>
<td>53</td>
<td>Worst</td>
</tr>
</tbody>
</table>

2017), and miniC2D (Oztok and Darwiche 2015). To accomplish this, we use a set of 1914 CNF literal-weighted model counting benchmarks, which were gathered from two sources.

First, the Bayes class\(^1\) contains 1091 benchmarks. The application domain is Bayesian inference (Sang, Beame, and Kautz 2005). The accompanied literal weights are in the interval [0, 1].

Second, the Non-Bayes class\(^2\) contains 823 benchmarks. This benchmark class is subdivided into eight families: Bounded Model Checking (BMC), Circuit, Configuration, Handmade, Planning, Quantitative Information Flow (QIF), Random, and Scheduling (Clarke et al. 2001; Sinz, Kaiser, and Küchlin 2003; Palacios and Geffner 2009; Klebanov, Manthey, and Muise 2013). All of these benchmarks are originally unweighted. As we focus in this work on weighted model counting, we generate weights by, for each variable \(x\), randomly assigning: either weights \(W^+(x) = 0.5\) and \(W^-(x) = 1.5\), or \(W^+(x) = 1.5\) and \(W^-(x) = 0.5\).\(^3\) Generating weights in this particular fashion results in a reasonably low amount of floating-point underflow and overflow for all model counters.

5.1 Experiment 1: Comparing ADDMC Heuristics

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 a single core, 24 GB of memory, and a 10-second timeout.

Performance Analysis Table 1 reports the numbers of benchmarks solved by four ADDMC heuristic configurations: best, second-best, median, and worst (of 245 configurations in total). Bouquet’s Method (BM) and bucket elimination (BE) have similar-performing top configurations: Best1 and Best2. This shows that Bouquet’s Method is competitive with bucket elimination.

See Figure 1 for a more detailed analysis of the runtime of these four heuristic configurations. Evidently, some configurations perform quite well while others perform quite poorly. The wide range of performance indicates that the choice of heuristics is essential to the competitiveness of ADDMC.

We choose Best1 (BM-Tree clustering with LexP cluster variable order and MCS diagram variable order), which was the heuristic configuration able to solve the most benchmarks within 10 seconds, as the representative ADDMC configuration for Experiment 2.

5.2 Experiment 2: Comparing Weighted Model Counters

In the previous experiment, 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 configuration, we now compare ADDMC to four state-of-the-art weighted model counters: Cachet, c2d\(^4\), d4, and miniC2D. (We note that Cachet uses long double precision, whereas all other model counters use double precision.)

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

Correctness Analysis To compare answers computed by different weighted model counters (in the presence of imprecision from floating-point arithmetic), we consider non-negative real numbers \(a \leq b\) equal when: \(b - a \leq 10^{-3}\) if \(a = 0\) or \(b \leq 1\), and \(b/a \leq 1 + 10^{-3}\) otherwise.

Even with this equality tolerance, weighted model counters still sometimes produce different answers for the same

\(^{1}\)https://www.cs.rochester.edu/u/kautz/Cachet/

\(^{2}\)http://www.cril.univ-artois.fr/KC/benchmarks.html

\(^{3}\)For each variable \(x\), 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 produced by Cachet on a formula \(\varphi\) by \(2^{\left|\mathcal{X}_S(\varphi)\right|}\) as a postprocessing step.

\(^{4}\)c2d does not natively support weighted model counting. To compare c2d to weighted model counters, we use c2d to compile CNF into d-DNNF then use 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.
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.

Performance Analysis Table 2 summarizes the performance of five weighted model counters (Cachet, ADDMC, miniC2D, c2d, and d4) as well as two virtual best solvers (VBS1 and VBS0) for various upper bounds on the MAVC. The MAVCs of the 1404 benchmarks solved by ADDMC within 1000 seconds range from 4 to 246.

Table 2: The numbers of benchmarks solved (of 1914 in total) by five weighted model counters and two virtual best solvers (VBS1 and VBS0).

<table>
<thead>
<tr>
<th>Solvers</th>
<th>Benchmarks solved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>VBS1 (with ADDMC)</td>
<td>–</td>
</tr>
<tr>
<td>VBS0 (without ADDMC)</td>
<td>–</td>
</tr>
<tr>
<td>d4</td>
<td>12</td>
</tr>
<tr>
<td>c2d</td>
<td>0</td>
</tr>
<tr>
<td>miniC2D</td>
<td>8</td>
</tr>
<tr>
<td>ADDMC</td>
<td>124</td>
</tr>
<tr>
<td>Cachet</td>
<td>14</td>
</tr>
</tbody>
</table>

See Figure 2 for a more detailed analysis of the runtime of all solvers. Evidently, VBS1 (with ADDMC) performs significantly better than VBS0 (without ADDMC). We conclude that ADDMC is a useful addition to the portfolio of weighted model counters.

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 100 seconds due to the large number of variables and clauses; these benchmarks were also not solved by ADDMC.

Figure 3 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 few benchmarks (12 of 379) with MAVCs greater than 100.

Figure 4 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.
In this work, 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 (Dechter 1999; Chavira and Darwiche 2007). On the other hand, Bouquet’s Method is much less well-known. Another direction for 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. There are other ADD packages that may be fruitful to explore in the future. For example, Sylvan (van Dijk and van de Pol 2015) supports multi-core operations on ADDs, which would allow us to investigate the impact of parallelism on our techniques. Moreover, Sylvan supports arbitrary-precision arithmetic.

Several other compact representations have been used in dynamic-programming frameworks for related problems. For example, AND/OR Multi-Valued Decision Diagrams (Mateescu, Dechter, and Marinescu 2008), Probabilistic Sentential Decision Diagrams (Shen, Choi, and Darwiche 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 Algebraic Decision Diagrams 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.

Acknowledgments
The authors would like to thank Dror Fried, Aditya A. Shrotri, and Lucas M. Tabajara for useful comments. This work was supported in part by the NSF (grants CNS-1338099, IIS-1527668, CCF-1704883, IIS-1830549, and DMS-1547433), by the Ken Kennedy Institute Computer Science & Engineering Enhancement Fellowship (funded by the Rice Oil & Gas HPC Conference), by the Ken Kennedy Institute for Information Technology 2017/2018 Cray Graduate Fellowship, and by Rice University.

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


Howard, R. A. 1966. Dynamic programming. JMS.


