A New Algorithm for Computing Upper Bounds for Functional E-MAJSAT

Knot Pipatsrisawat and Adnan Darwiche
{thammakn,darwiche}@cs.ucla.edu

Computer Science Department
University of California, Los Angeles

Abstract. We present a new method for computing upper bounds for an optimization version of the E-MAJSAT problem. This new approach is based on the use of the compilation language d-DNNF that underlies several state-of-the-art algorithms for solving problems related to E-MAJSAT. We show that the new bound values dominate those produced by the standard algorithm based on the same approach. Moreover, we present a technique for pruning values from the branch-and-bound search tree based on the information available after each bound computation. We integrate our new techniques into a probabilistic conformant planner ComPlan and demonstrate significant empirical improvements.

1 Introduction

Many real-world problems require uncertainty in their formulations. As a result, many researchers have studied methods for modeling and solving problems that exhibit uncertainty. One prototypical formulation of such problems is the E-MAJSAT problem [15], which asks whether there exists an assignment to a given set of variables of a given formula such that the majority of assignments to the remaining variables satisfy the formula.

E-MAJSAT, which is $\mathrm{NP}^{\mathrm{PP}}$-complete [15, 18], is an extension of the boolean satisfiability (SAT) problem that includes an element of model counting into its formulation.[1] As a result, it can be used to model many interesting problems in AI such as probabilistic conformant planning [8, 9, 13], finding maximum a posteriori hypothesis (MAP) [20], and finding maximum expected utility (MEU) solution [7]. Intuitively, to solve an E-MAJSAT problem, we need to search in an exponential search space (the NP part), while checking whether each candidate constitutes a solution requires solving a counting problem (the PP part). E-MAJSAT (and its different formulations) is an important problem in AI and has been studied extensively in the literature [15, 18, 20, 7].

Many exact algorithms for solving this class of problems have been proposed in the past. For example, in [14, 16], the authors proposed a modified version of the DPLL algorithm [6] for solving E-MAJSAT, Dechter *et al* [7] used bucket

[1] E-MAJSAT is a special case of a more general class of problems called the stochastic satisfiability problem [16].

elimination for solving MAP, while recursive conditioning was used for solving the same problem in [2]. In this work, we investigate an algorithm for computing upper bounds on the solution of an optimization version of E-MAJSAT. Such an algorithm is useful as it can be employed by a branch-and-bound search algorithm for solving the problem.

The proposed algorithm can be viewed as an improvement of those used in [11, 10], which take advantage of a compilation language called d-DNNF for computing upper bounds. In this work, we point out the cause of bound looseness in these existing algorithms and propose a method for reducing the inaccuracy in bound values. We show that new bounds generated by our algorithm are at least as tight as those produced by the aforementioned work. Moreover, we describe a method of using information available from the new algorithm to dynamically prune branches of the search tree of the branch-and-bound search algorithm. We tested our techniques by integrating it into a branch-and-bound probabilistic conformant planner, ComPlan [10]. Empirical results show significant improvements after the integration.

The rest of the paper is organized as follows. We first discuss, in Section 2, basic notations and definitions that will be used in future discussions. Then, in Sections 3 and 4, we review existing techniques for solving and computing bounds of an optimization version of E-MAJSAT based on d-DNNF. We present a new algorithm for computing tighter bounds in Section 5 and discuss some of its properties. Section 6 discusses its integration with a branch-and-bound solver and presents a technique for pruning search branches based on information available after each bound computation. Experimental results are presented in Section 7 and we conclude in Section 8.

2 Background

We present, in this section, some basic notations that will be used throughout the paper. Given a literal ℓ, which is either a variable or the negation of a variable, we use $var(\ell)$ to refer to its variable. An *assignment* is simply a consistent set of literals (interpreted as their conjunction). If Δ is a propositional sentence and $\mathbf{s}$ is an assignment, we say that $\mathbf{s} \models \Delta$ iff $\mathbf{s}$ satisfies Δ. Moreover, we use $\Delta|\mathbf{s}$ to denote the formula obtained from Δ by substituting every literal $\ell \in \mathbf{s}$ with **true** and every literal ℓ such that $\neg\ell \in \mathbf{s}$ with **false**.

Unless stated otherwise, in this paper, we will assume that every variable of a given formula has been designated as either a *choice variable* or a *chance variable*. Moreover, we assume that the probabilities θ of the chance variables are given. We use $\theta(r)$ to denote the probability of a chance literal r. If Ψ is a formula containing only chance variables, then we define the *probability* of Ψ to be

$$\mathsf{Pr}(\Psi) = \sum_{\mathbf{r} \models \Psi} \left(\prod_{r \in \mathbf{r}} \theta(r) \right).$$

The summation above is over all complete assignments of the chance variables that satisfy Ψ. The probability of Ψ is essentially the weighted model count of Ψ, where the weight of each model is simply the product of the probabilities of its chance literals.[2]

In this work, we investigate an optimization version of the E-MAJSAT problem [15], which we call functional E-MAJSAT. Given a CNF Γ, the functional E-MAJSAT problem on Γ is the problem of finding the maximum probability of Γ under any complete assignment $\mathbf{e}$ of the choice variables.[3] More formally, we want to compute

$$M = \max_{\mathbf{e}} \mathsf{Pr}(\Gamma|\mathbf{e})$$

We will refer to M as the *maximum probability* of the functional E-MAJSAT problem on Γ.

3 Solving Functional E-MAJSAT via Knowledge Compilation

Given a functional E-MAJSAT problem, one way of computing the solution to the problem is to convert the CNF formula into deterministic decomposable negation normal form (d-DNNF) (to be defined next). If the conversion is performed in a certain way (to be discussed), we can obtain the maximum probability of the functional E-MAJSAT problem by a linear traversal of the d-DNNF.

3.1 Deterministic Decomposable Negation Normal Form (d-DNNF)

A negation normal form (NNF) is a rooted directed acyclic graph (DAG) where each leaf is labeled with **true**, **false**, or a literal, and each internal node is labeled with either AND or OR. Each node in the DAG is used to represent the formula rooted at that node. Deterministic decomposable negation normal form (d-DNNF) [5] is a subset of NNF that satisfies *determinism* and *decomposability*. An NNF is said to be *deterministic* iff, for each OR, no two children share a model, and is said to be *decomposable* iff, for each AND, no two children share a variable. Figure 1 depicts a d-DNNF (we will explain the letters and numbers in parentheses later).

For simplicity, we will assume that every OR node has exactly two children, while an AND node can have any number of children. In this work, we use C2D [4, 1] to compile CNF formulas into d-DNNF. C2D produces d-DNNF with the following key property: every OR node (which has to be deterministic) is of the form $\alpha = (x \wedge \beta) \vee (\neg x \wedge \gamma)$. Here, x is a variable which is called the

[2] Note that this definition does not require Ψ to be any particular form.
[3] $\Gamma|\mathbf{e}$ is a formula containing only chance variables.

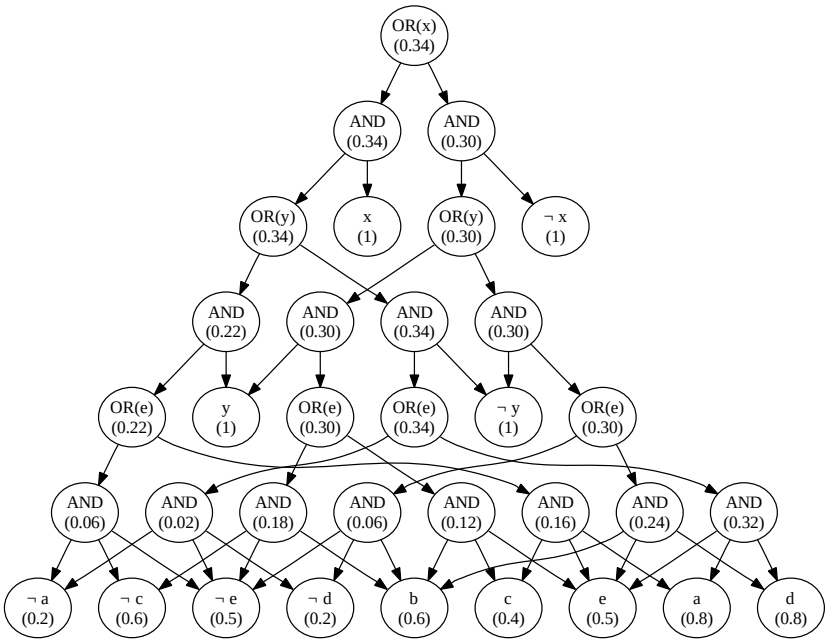

Fig. 1. A d-DNNF. In this graph, no two children of any AND node share a variable and no two children of any OR node share a model. Each OR node is annotated with its decision variable. Moreover, each node's value is shown in parentheses.

decision variable of α, denoted $dec(\alpha)$. For the rest of the paper, we will assume that every d-DNNF has this special form. The information about the decision variable of each OR node is available in the output of C2D and will be used by algorithms described later. The decision variable of each OR node is shown in parentheses for the d-DNNF in Figure 1.

3.2 Solving Functional E-MAJSAT Exactly

In this section, we will present an existing algorithm for solving functional E-MAJSAT based on a linear traversal of d-DNNF. This algorithm forms a basis for bound computation methods to be discussed in later sections.

As hinted earlier, not every d-DNNF can be used to solve functional E-MAJSAT exactly this way. A d-DNNF is said to be *constrained* if at most one value (literal) of each choice variable appears below any OR node with a chance decision variable. Consider the d-DNNF in Figure 1 again. If we let x, y be the only choice variables, then this d-DNNF is constrained. In particular, neither x nor y appears below any OR node with a chance decision variable (all of which are at depth 4). Given a constrained d-DNNF, we can solve functional E-MAJSAT exactly by a single traveral of the graph [11]. The d-DNNF in Figure 1 is actually equivalent to the following CNF:

$$(x \vee b \vee e) \wedge (x \vee b \vee \neg e) \wedge (\neg x \vee a \vee \neg e) \wedge (\neg x \vee \neg a \vee e) \wedge$$
$$(y \vee d \vee \neg e) \wedge (y \vee \neg d \vee e) \wedge (\neg y \vee c \vee \neg e) \wedge (\neg y \vee \neg c \vee e)$$

Therefore, this d-DNNF can be used to efficiently solve the functional E-MAJSAT problem on this CNF formula.

In general, given a CNF Γ, a corresponding constrained d-DNNF Δ, and an assignment $\mathbf{s}$ (which could be partial or empty) to the choice variables, we can solve the functional E-MAJSAT problem on the formula $\Gamma|\mathbf{s}$ as follows. We simply perform a bottom-up traversal of Δ and, for each node α, we compute its value, $val(\alpha, \mathbf{s})$, which is defined recursively as

$$val(\alpha, \mathbf{s}) = \begin{cases} \theta(r), & \text{if } \alpha = r \text{ is a chance literal} \\ 0, & \text{if } \alpha \text{ is a choice literal and } \mathbf{s} \models \neg\alpha \\ 1, & \text{if } \alpha \text{ is a choice literal and } \mathbf{s} \not\models \neg\alpha \\ \prod_i val(\alpha_i), & \text{if } \alpha = \bigwedge_i \alpha_i \\ val(\alpha_1) + val(\alpha_2), & \text{if } \alpha = \alpha_1 \vee \alpha_2 \text{ and} \\ & dec(\alpha) \text{ is a chance variable.} \\ \max(val(\alpha_1), val(\alpha_2)), & \text{if } \alpha = \alpha_1 \vee \alpha_2 \text{ and} \\ & dec(\alpha) \text{ is a choice variable.} \end{cases} \quad (1)$$

The maximum probability of this problem is simply the value of the root node.

Consider the example in Figure 1 again. We set $\theta(a) = 0.8, \theta(b) = 0.6, \theta(c) = 0.4, \theta(d) = 0.8, \theta(e) = 0.5$. In this case, the maximum probability of this functional E-MAJSAT problem is $val(\Delta, \mathbf{true}) = 0.34$. The number in parentheses at each node α is simply the value of α under no assignment to x and y.

The following proposition states a guarantee on the value computed from a constrained d-DNNF.[4] Due to space constraint, we present a proof of this result in a separate report [21].

Proposition 1. *Given a CNF Γ and a constrained d-DNNF Δ which is equivalent to Γ, $val(\Delta, \mathbf{true})$ is the maximum probability of the functional E-MAJSAT problem on Γ.*

According to this proposition, whenever the d-DNNF is constrained, the value of the root as defined in Equation 1 always corresponds to an actual $\mathsf{Pr}(\Gamma|\mathbf{e})$ of some complete assignment to the choice variables $\mathbf{e}$. Moreover, this value is guarantee to be the maximum probability of the formula under any complete assignment of the choice variables.

In general, the time (and space) complexity of compiling CNF into a constrained d-DNNF is exponential in the *constrained treewidth* of the CNF [19].[5] This constraint could render compilation very impractical [18]. Nevertheless, if

[4] A similar claim was made in [11] without a proof.

[5] The constrained treewidth is the treewidth obtained from an elimination order in which all the choice variables are eliminated last.

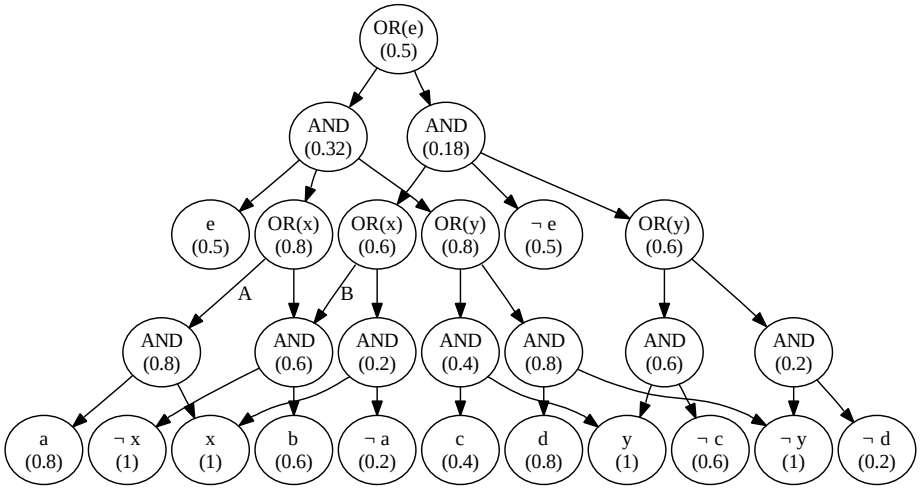

Fig. 2. A d-DNNF which is compiled without any constraint. Each node is labeled with the value used for bound computation.

we disregard this constraint, the compilation will only be exponential in the treewidth. The resulting d-DNNF can then be used to compute an upper bound on the maximum probability of the problem. We discuss this approach in the next section.

4 Functional E-MAJSAT Bound Computation from d-DNNF

Consider Figure 2 which shows a d-DNNF that is logically equivalent to the one in Figure 1 but is not constrained. The nodes are labeled with values defined by Equation (1). If we assume that there is currently no assignment to the choice variables (x and y), then the value of the root of this d-DNNF is 0.5, which is greater than the actual maximum probability of this problem.

In general, if the d-DNNF is not constrained, the value of the root will only be an upper bound on the maximum probability of the functional E-MAJSAT problem [11]. In a special case when the assignment $\mathbf{s}$ contains all choice variables, $val(\Delta, \mathbf{s})$ simply becomes $\mathsf{Pr}(\Delta|\mathbf{s})$, the weighted model count of $\Delta|\mathbf{s}$. Even though we lose the ability to compute the exact maximum probability by ignoring the constraint, one advantage of this approach is that the compilation process now becomes exponential in only the (unconstrained) treewidth of the CNF formula [3]. The difference between the constrained and unconstrained treewidth could be significant for some problems [18]. Examples of algorithms that utilize this bound computation approach are [11] (for solving MAP) and [10] (for probabilistic planning).

4.1 The Cause of Bound Looseness

As shown earlier, whenever we have only a partial assignment to the choice variables, the value computed from an unconstrained d-DNNF may overestimate the maximum probability of the problem. The reason that the computed value may be too large is because the algorithm allows unrealistic assignments to the choice variables to be considered. In particular, the free choice variables can be "assigned" both the values **true** and **false** at the same time, during the traversal of unconstrained d-DNNF. This behavior is caused by the fact that every node labeled with a literal of a free choice literal has value 1 (the third case of Equation 1). This reflects the algorithm's inability to determine which value of each free choice variable is the best assignment. As a result, the value of the root node may not correspond to the probability of the formula conditioned by any complete assignment to the choice variables. In the example in Figure 2, we can see that the top left AND node attains the value of 0.32 by selecting the value **true** for x (indicated by the edge labeled with "A"). However, the top right AND node attains the value of 0.18 by selecting the value **false** for x (indicated by the edge labeled with "B"). As a result, the value 0.5 at the root cannot be realized by any valid assignment to x and y (i.e. x cannot be **true** and **false** at the same time). In the next section, we introduce a way of mitigating this problem by keeping track of more information as we traverse the d-DNNF.

5 Computing Tighter Bounds Using Option Pairs

In this section, we discuss an algorithm for reducing the looseness of bounds obtained from unconstrained d-DNNF. The key idea is to compute bound values that are conditioned on the values of choice variables. To do so, we will need to make more information available at each node of the d-DNNF. The following definition is needed in the discussion of our algorithm.

Definition 1 (Option Pair). *Given a d-DNNF node α, a partial instantiation of the choice variables $\mathbf{s}$ and a choice variable x not mentioned by $\mathbf{s}$, $\psi = (x, v^+, v^-)$ is an option pair of α on x if*

- *v^+ is an upper bound of $val(\alpha, x \wedge \mathbf{s})$ and*
- *v^- is an upper bound of $val(\alpha, \neg x \wedge \mathbf{s})$.*

In this case, x is called the option variable, denoted $v(\psi)$. We will also call v^+ the positive option ($p(\psi)$) and call v^- the negative option ($n(\psi)$) of ψ. The best option of ψ is simply $\max(v^+, v^-)$.

An option pair contains bounds on the node's values conditioned on the values of a choice variable. In our new bound computation algorithm, instead of computing just one value for each node in the d-DNNF, we compute option pairs on free choice variables that appears below the node (if they exist). Not every node in a d-DNNF can have an option pair. Particularly, if a node does not contain any free choice variable below it, it will not have any option pair

defined. On the other hand, if a node mentions multiple free choice variables, it will have more than one option pair. Before we describe an algorithm for computing option pairs, we need some definitions for the value of an option pair and the value of a node, under an assignment. Given an assignment $\mathbf{s}$ to the choice variables, the *contribution* of an option pair ψ is the largest option value it can contribute under the assignment. This is defined formally as

$$\kappa(\psi, \mathbf{s}) = \begin{cases} p(\psi), & \text{if } \mathbf{s} \models v(\psi) \\ n(\psi), & \text{if } \mathbf{s} \models \neg v(\psi) \\ \max(p(\psi), n(\psi)), & \text{otherwise.} \end{cases}$$

In light of this definition, we can redefine the value of each node. If a node α has at least one option pair, we define its value $val^\star(\alpha, \mathbf{s})$ to be the smallest contribution from any option pair of the node. This is the tightest bound we can put on the value of the node, because any complete assignment must set every choice variable to a value. If the node does not have an option pair, its value is defined to be $val(\alpha, \mathbf{s})$ (as defined in Equation (1)). Formally, we have

$$val^\star(\alpha, \mathbf{s}) = \begin{cases} \min_\psi \kappa(\psi, \mathbf{s}), & \alpha \text{ has some option pairs} \\ val(\alpha, \mathbf{s}), & \text{otherwise.} \end{cases}$$

Given a node α that mentions at least one free choice variable V, we define the option pair of α on V (under assignment $\mathbf{s}$) as follows.

1. If $\alpha = \ell$ (ℓ must be a literal of V), its option pair on V is $(V, \theta(\ell), 0)$ if ℓ is positive, and $(V, 0, \theta(\ell))$ otherwise.
2. If $\alpha = \bigwedge_{i=1}^{n} \alpha_i$,[6] its option pair on V is
$$(V, \prod_{i=1}^{n} val^\star(\alpha_i, \mathbf{s} \wedge V), \prod_{i=1}^{n} val^\star(\alpha_i, \mathbf{s} \wedge \neg V)).$$
3. If $\alpha = \alpha_1 \vee \alpha_2$ and $dec(\alpha)$ is a choice variable, its option pair on V is
$$(V, \max(val^\star(\alpha_1, \mathbf{s} \wedge V), val^\star(\alpha_2, \mathbf{s} \wedge V)), \max(val^\star(\alpha_1, \mathbf{s} \wedge \neg V), val^\star(\alpha_2, \mathbf{s} \wedge \neg V))).$$
4. If $\alpha = \alpha_1 \vee \alpha_2$ and $dec(\alpha)$ is a chance variable, its option pair on V is
$$(V, val^\star(\alpha_1, \mathbf{s} \wedge V) + val^\star(\alpha_2, \mathbf{s} \wedge V), val^\star(\alpha_1, \mathbf{s} \wedge \neg V) + val^\star(\alpha_2, \mathbf{s} \wedge \neg V)).$$

This algorithm can be repeated at each node on every free choice variable to compute all option pairs. The bound value produced by this algorithm is then the value of the root node ($val^\star(\Delta, \mathbf{s})$). Although computing option pairs for all free choice variables at each node could lead to much tighter bounds, its time complexity is $O(|\mathbf{E}||\Delta|)$, where $|\mathbf{E}|$ is the number of choice variables and $|\Delta|$ is the size of the d-DNNF.[7] In practice, any number of option pairs can

[6] In this case (AND), the set of option variables of the children are disjoint.

[7] In practice, $val^\star(\alpha, \mathbf{s} \wedge \ell)$ is simply the minimum of $val^\star(\alpha, \mathbf{s})$ and the option corresponding to ℓ of α. Therefore, computing $val^\star(\alpha, \mathbf{s} \wedge \ell)$ is a constant-time operation, given that $val^\star(\alpha, \mathbf{s})$ is stored at each node.

be computed at each node, but one needs a heuristic for selecting which option pairs to compute.

The source of improvement of this new bound algorithm lies in Step 4 above. In this case, instead of simply adding the highest values that each child may produce together, we only add values that have compatible underlying assignments of the option variables together. The value of such an OR node under the new bound algorithm will be lower than the one computed the previous algorithm (without option pairs) whenever the best options of the children correspond to conflicting assignments of some option variable. Therefore, the quality of the bounds generated by option pairs is more sensitive to changes in the parameters of the problems (i.e. the probabilities of the chance literals). In any case, we have the following results regarding the bounds computed this way.

Proposition 2 (Bound Correctness). *Given a d-DNNF Δ and an instantiation (possibly partial) $\mathbf{s}$ of choice variables, we have*

$$val^\star(\Delta, \mathbf{s}) \geq \max_{\mathbf{x}} \mathsf{Pr}(\Delta|(\mathbf{s} \wedge \mathbf{x})),$$

where $\mathbf{x}$ is a complete assignment of the remaining choice variables.

Proposition 3 (Dominance). *Given a d-DNNF Δ and an instantiation $\mathbf{s}$ of choice variables, we have $val^\star(\Delta, \mathbf{s}) \leq val(\Delta, \mathbf{s})$.*

These results show that the values computed using option pairs are correct upper bounds that are at least as tight as those computed using the basic algorithm in the previous section.

Let us now illustrate the new bound algorithm with an example. Consider again the d-DNNF in Figure 2. In the previous section, we showed a basic bound computation which resulted in the bound value of 0.5 at the root. Using option pairs on this d-DNNF, we obtain Figure 3. In this example, there are still two choice variables, x, y. We will now explain the computation of option pairs of some nodes in this d-DNNF.

Each leaf node labeled with a chance variable in this figure is only annotated with its value, because it has no option pairs. Other leaves are labeled with option pairs on their choice variables. Every OR node with a choice decision variable (all at depth 2) mentions only one choice variable (which is its decision variable). Their option pairs are obtained by simply combining the best option from each child. Each of the AND nodes at depth 1 mentions two choice variables. For the option pair on x, the positive option is obtained by multiplying the positive option of the child that mentions x with the values of the remaining children (which do not mention x). The negative option and the option pair on y can be computed in a similar way. To compute the option pair at the root node, which is an OR node with a chance decision variable, we simply add the compatible options of the children together. In the case of x, the option pair of the left child indicates that if $x = \mathbf{true}$, its value is no more than 0.32, while the right child indicates that its value is no more than 0.06 under the same assignment.

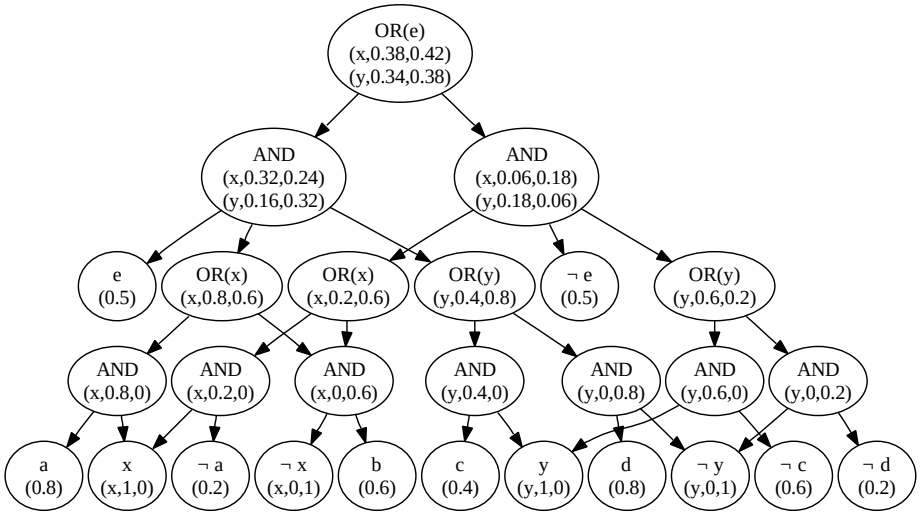

Fig. 3. Bound computation on a d-DNNF using option pairs.

As a result, we can conclude that, if $x = \mathbf{true}$, the root's value can be no more than 0.38. The other option values at the root can be computed in a similar way. Finally, from this bound computation, we can conclude that, no matter what value is assigned to y, the value of the root cannot be larger than 0.38.[8] Hence, we have obtained a bound value that is tighter than that computed by the algorithm in the previous section.

In the next section, we will present a technique that leverages on the option pairs at the root node to speed up branch-and-bound search.

6 Utilizing the New Bound Computation in a Branch-and-Bound Algorithm

One natural application of our new bound computation algorithm is in a branch-and-bound search algorithm for solving functional E-MAJSAT. Many algorithms based on branch-and-bound search have been proposed previously, for example [14, 16, 19, 11, 10]. The algorithms search in the space of all possible assignments to the choice variables. Each leaf of this search tree corresponds to a complete assignment $\mathbf{e}$ of the choice variables and is associated with the probability $\mathsf{Pr}(\Delta|\mathbf{e})$. Solving functional E-MAJSAT is then equivalent to finding the leaf with the highest probability.

At any given point in time, the solver keeps track of the highest probability associated with any complete assignment seen so far. This value is a lower bound

[8] The same analysis on x yields a looser bound of 0.42. We takes the smaller value as any complete assignment must set every choice variable to a value.

(LB) of the maximum probability for the problem. Then, at each internal node of the search tree, the solver computes an upper bound (UB) of the probability of any leaf below the node. Whenever UB is less than or equal to to current value of LB, the search tree below the current node can be pruned, as it means that no better assignment exists in that part of the search space. Algorithm 1 shows the pseudo-code of a branch-and-bound algorithm for functional E-MAJSAT. This algorithm assumes that LB is initialized to 0 and *solution* to null.

Algorithm 1: BnB-fEmajSAT

global : CNF Γ, a set of choice variables $\mathbf{E}$, LB, *solution*
input : A set of assignments to choice variables $\mathbf{s}$
output: An assignment with the maximum probability is stored in *solution* and its probability in LB.

```
1  if |s| ≠ |E| then
2      select a free choice variable X
3      for each value x of X do
4          if bound(Γ, s ∧ x) > LB then
5              BnB-fEmajSAT(s ∧ x)
6  else
7      p = Pr(Γ|s)
8      if p > LB then
9          LB = p
10         solution = s
```

To integrate the new bound computation technique into this algorithm, we need to

1. compile Γ into a d-DNNF Δ before the first call to BnB-fEmajSAT.
2. replace $bound(\Gamma, \mathbf{s})$ and $\mathsf{Pr}(\Gamma|\mathbf{s})$ (Lines 4 and 7) by $val^\star(\Delta, \mathbf{s})$.

These changes do not compromise the correctness of the algorithm, because $val^\star$ always produces an upper bound to the problem (Proposition 2).

6.1 Using Option Pairs for Pruning Values

When we use multiple option pairs for computing bounds, we usually have a number of option pairs available at the root of the d-DNNF, even though only one bound value (the smallest best option) is extracted from this process.

Consider an example situation where we have the following option pairs at the root: $(x, 0.75, 0.85), (y, 0.9, 0.7), (z, 0.5, 0.88), (w, 0.81, 0.88)$. Clearly, the current bound value is 0.85. Let us assume further that the current value of LB is 0.8. At this point, we cannot prune the search tree yet, because UB$\not\leq$LB. However, we know that if we set x to **true**, the bound value will suddenly be smaller than LB. The same can be concluded about setting $y =$ **false**, and $z =$ **true**.

Therefore, without any additional work, we can prune the following values from the current sub-tree: $x = \textbf{true}, y = \textbf{false}, z = \textbf{true}$.

In general, after computing a bound using option pairs, we inspect each option pair ψ of the root node, if $p(\psi) \leq LB$, then the branch $v(\psi) = \textbf{true}$ can be removed. If $n(\psi) \leq LB$, then the branch $v(\psi) = \textbf{false}$ can be removed.[9] During this process, the value of each choice variable can be pruned independently, no matter what values of other variables are pruned. This process of computing a bound and removing values can be repeated as long as a new value is removed. Once no new value can be pruned, we can continue to search by branching on a free choice variable. This algorithm for pruning values can be easily incorporated into Algorithm 1 right after the bound computation on Line 4.

7 Experimental Results

The bound computation algorithm presented here, along with the technique that utilizes option pairs to prune values, can be integrated into any algorithm that employs the bound computation algorithm discussed in Section 4. Next, we briefly discuss our integration of the techniques into a probabilistic conformant planner, ComPlan [10], which is a state-of-the-art planner of its kind.

7.1 Integration with ComPlan

Probabilistic conformant planning is a type of planning which allows uncertainty in both the initial state and the outcomes of actions. ComPlan is a branch-and-bound probabilistic conformant planner which finds a plan with the maximum success probability for a given plan length. It utilizes d-DNNF (without option pairs) for bound computation as discussed in Section 4. First, ComPlan converts each planning problem into a CNF formula, then it compiles the formula into d-DNNF using C2D. At each search node, besides computing a bound from d-DNNF, it also prunes values by trying each individual value of the current free choice variables and measuring the bound afterwards; a value can be pruned if it yields a bound that is $\leq LB$. ComPlan also uses a dynamic variable and value ordering heuristic based on the bound values computed during value pruning.

We implemented the planner based on the descriptions in [10].[10] Then, we modified the bound computation algorithm to keep track of all option pairs at each node. We also replaced ComPlan's value pruning algorithm with the one that is based on option pairs (Section 6.1). We call our version of the planner ComPlan+. Currently, ComPlan+ only uses a static variable and value ordering heuristic, which is described in [11].

[9] If both values of a variable are pruned, the search algorithm can backtrack.

[10] Although our implementation of ComPlan does not behave exactly like the one presented in [10] (e.g. number of nodes visited), its performance is comparable (based on the running time reported in that paper).

7.2 Results on Planning Problems

We compared the performance of ComPlan and ComPlan+ on different domains of probabilistic conformant planning problems. In particular, we consider the domains sand-castle [17] and slippery-gripper [13] (as extended by [12]). Each of these domains contains problems of finding a plan with the highest success probability for the given horizon (number of actions).[11] All of the experiments were conducted on a Pentium 4, 3.8GHz machine with 4GB of RAM.

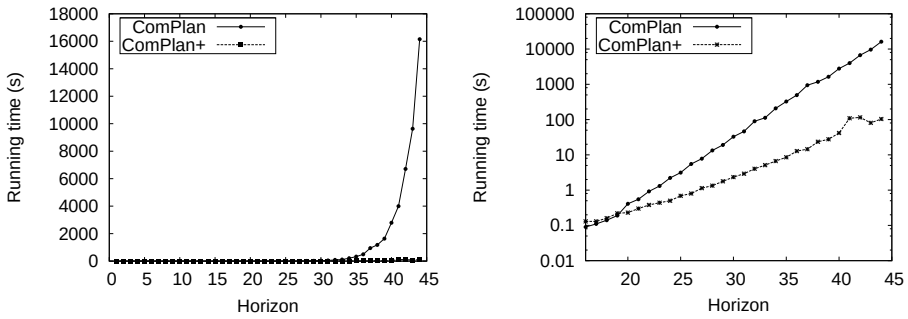

Fig. 4. Running time of ComPlan and ComPlan+ on sand-castle. The results are plotted on normal running time scale (left) and on log-scale (right).

Figure 4 shows plots of both planners' running time on sand-castle. On the left plot, the data are shown on a linearly-scaled y-axis. On the right plot, the y-axis is log-scaled (with the same data). We can clearly see from these plots that ComPlan+ significantly outperforms ComPlan. Moreover, the running time ratio (ComPlan over ComPlan+) increases as the horizon grows. When the horizon is equal to 44, ComPlan takes 16,152 seconds to solve the problem, while ComPlan+ only takes 104 seconds. In our experiment (result not shown in the plots), ComPlan+ can solve the problem with horizon equals to 50 in less than 700 seconds, while ComPlan does not finish after a day.

Figure 5 shows plots of running time of the planners on slippery-gripper. There are two sets of slippery-gripper problems in our experiment. The first set is exactly the slippery-gripper problems described in [12]. The results on this set is on the left plot of the figure. The other set of problems, whose results are shown on the right of Figure 5, contains modified slippery-gripper problems, which differ from the original ones only in the probabilities of success of some actions. In particular, we changed the probability of the action DRY being successful from 0.8 to 0.9, changed the probability that the action PAINT will make the gripper which is not holding the block dirty from 0.1 down to 0.05, and changed the probability of success of the action PICKUP when the gripper is wet from 0.5 to

[11] In these planning problems, the variables that represent plans are the choice variables and those that represent uncertainty in the initial state or action outcomes are the chance variables.

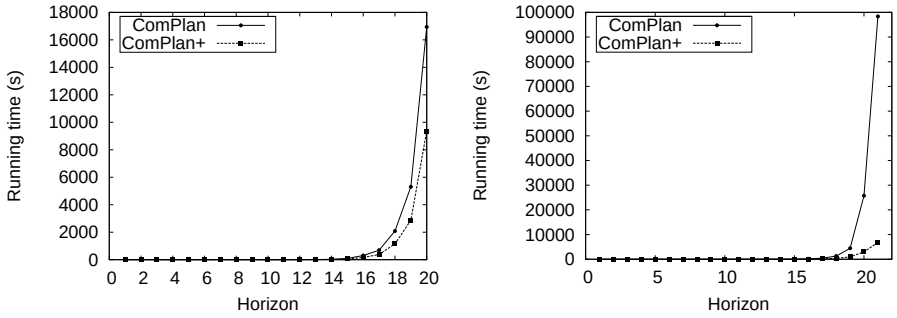

Fig. 5. Running time of ComPlan and ComPlan+ on (left) original slippery-gripper and (right) modified slippery-gripper.

0.85. This modification makes the actions slightly more deterministic, exposing the drawbacks of the basic bound computation approach more.

According to Figure 5, ComPlan+ exhibits a constant factor (about 1.8) improvement over ComPlan on the original set of slippery-gripper problems. However, when we modified the probabilities of the problems, the difference between ComPlan and ComPlan+ becomes greater. When the horizon is 20, ComPlan takes 25,745 seconds, while ComPlan uses only 2,992 seconds. When the horizon is 21, ComPlan takes 98,359 seconds, while ComPlan+ takes 6,753 seconds. Similar to the results for sand-castle, the running time ratio between the two planners increases as the problem's difficulty increases. The changes in parameters simply create more conflicting values of choice variables during bound computations, which lead to greater differences between the normal bounds and those computed using option pairs.

8 Conclusions

In this paper, we proposed a new bound computation method, based on compilation to d-DNNF, for the functional E-MAJSAT problem. The algorithm can be used for computing bounds in a branch-and-bound solver for functional E-MAJSAT. In addition to yielding tighter bounds, the new algorithm also produces additional information that allows the solver to prune values as it searches for the best solution at virtually no additional cost. We integrated the new techniques into a branch-and-bound probabilistic conformant planner, ComPlan, and showed empirically that the new techniques yield significant improvement, which, on some problem domains, grows as the problem size increases.

Acknowledgment

The authors would like to thank Jinbo Huang and Mark Chavira for useful discussions and for answering questions about their solvers.

References

1. Darwiche, A. The c2d compiler. Available at http://reasoning.cs.ucla.edu/c2d/.
2. Darwiche, A. Any-space probabilistic inference. In *Proceedings of UAI-00* (San Francisco, CA, 2000), Morgan Kaufmann, pp. 133–1.
3. Darwiche, A. On the tractability of counting theory models and its application to belief revision and truth maintenance. *JANCL 11*, 1-2 (2001), 11–34.
4. Darwiche, A. New advances in compiling CNF to decomposable negational normal form. In *Proceedings of ECAI-04* (2004), pp. 328–332.
5. Darwiche, A., and Marquis, P. A knowledge compilation map. *Journal of Artificial Intelligence Research 17* (2002), 229–264.
6. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (1962), 394–397.
7. Dechter, R. Bucket elimination: a unifying framework for probabilistic inference. In *Proceedings of UAI-96* (1996), pp. 211–219.
8. Drummond, M., and Bresina, J. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of AAAI-90* (1990), pp. 138–144.
9. Hanks, S. *Projecting plans about undertain worlds*. PhD thesis, 1990.
10. Huang, J. Combining knowledge compilation and search for conformant probabilistic planning. In *Proceedings of ICAPS-06* (2006), pp. 253–262.
11. Huang, J., Chavira, M., and Darwiche, A. Solving map exactly by searching on compiled arithmetic circuits. In *Proceedings of AAAI-06* (2006), pp. 143–148.
12. Hyafil, N., and Bacchus, F. Conformant probabilistic planning via csps. In *ICAPS* (2003), pp. 205–214.
13. Kushmerick, N., Hanks, S., and Weld, D. S. An algorithm for probabilistic planning. *Artificial Intelligence 76*, 1-2 (1995), 239–286.
14. Littman, M. L. Initial experiments in stochastic satisfiability. In *AAAI '99/IAAI '99* (1999), pp. 667–672.
15. Littman, M. L., Goldsmith, J., and Mundhenk, M. The computational complexity of probabilistic planning. *JAIR 9* (1998), 1–36.
16. Littman, M. L., Majercik, S. M., and Pitassi, T. Stochastic boolean satisfiability. *J. Autom. Reason. 27*, 3 (2001), 251–296.
17. Majercik, S. M., and Littman, M. L. Maxplan: A new approach to probabilistic planning. In *AIPS* (1998), pp. 86–93.
18. Park, J. Map complexity results and approximation methods. In *Proceedings of UAI-02* (2002), pp. 388–396.
19. Park, J., and Darwiche, A. Solving map exactly using systematic search. In *Proceedings of UAI-03* (2003), pp. 459–468.
20. Park, J., and Darwiche, A. Complexity results and approximation strategies for map explanations. *Journal of Artificial Intelligence Research 21* (2004), 101–133.
21. Pipatsrisawat, K., and Darwiche, A. A new algorithm for computing upper bounds for functional E-MAJSAT. Tech. Rep. D–156, Automated Reasoning Group, Computer Science Department, UCLA, 2008.