On the Impact of Junction-Tree Topology on Weighted Model Counting

Batya Kenig and Avigdor Gal

Technion, Israel Instutute of Technology, Haifa, Israel

Abstract. We present and evaluate the power of a new framework for weighted model counting and inference in graphical models, based on exploiting the topology of the junction tree representing the formula. The proposed approach uses the junction tree topology in order to craft a reduced set of partial assignments that are guaranteed to decompose the formula. We show that taking advantage of the junction tree structure, along with existing optimization methods borrowed from the CNF-SAT domain, can translate into significant time savings for weighted model counting algorithms.

1 Introduction

Weighted Model Counting (WMC) on a propositional knowledge base in Conjunctive Normal Form (CNF) is an effective and popular approach to solve problems of exact probabilistic inference [23, 1, 4], conformant planning [11], and the study of hard combinatorial problems [12] by taking advantage of local structures. WMC is based on the model counting or #SAT problem [12], where the objective is to count the number of assignments that satisfy the propositional formula. WMC generalizes model counting by assigning a weight to each literal, and computing the weighted sum of satisfying assignments.

Model counting (and WMC) is #P-hard in general [25]. However, much work is devoted to create methods that capitalize on local structure in the form of determinism and context specific independence to enable significant speedups compared to classic inference approaches [9, 18].

In this work we continue this line of research and propose a novel approach for performing WMC that is based on message passing in junction trees. We observe that the topology of a formula's junction tree reveals structure that can be utilized for enhancing the performance of WMC. The algorithm we propose in this work generates compact factors that contain a small set of mutual exclusive and exhaustive partial assignments that are guaranteed to decompose the formula.

We evaluate the proposed approach on three benchmarks, comparing it to `c2d` [6], a leading compiler for WMC. The empirical analysis leads to interesting observations about the pros and cons of each of the methods.

The rest of the paper is organized as follows. Junction trees are introduced in Sec. 2 followed by the introduction of CNF-trees and their role in modeling the underlying conditional independences between formula variables (Section 3). Next, we outline the

main idea of the paper, where formula decomposition is performed by partial assignments (Sec. 4). We show how to generate the reduced set of partial assignments (modeled as tree-CPTs) in Sec. 5. Sec. 6 presents the empirical evaluation. We conclude with a discussion of related work (Sec. 7) and concluding remarks (Sec. 8).

2 Background: The Junction Tree Algorithm

In what follows, we denote variables in upper case letters (*e.g.*, X) and their instantiations in lower case (*e.g.*, x). Sets of variables are denoted using bold upper case letters (*e.g.*, $\mathbf{X}$) and their instantiations in bold lower case letters (*e.g.*, $\mathbf{x}$).

A Probabilistic Graphical Model (PGM) is a graph $G(V, E)$ in which nodes represent random variables $\mathbf{X} = \{X_i : i \in V\}$, and edges represent direct dependencies between them. The graphical model contains a set of discrete functions $\mathbf{F}$, termed *factors*, that are defined over a subset of its variables. Factors are typically represented as tables, indexed by variable instantiations. Formally, a factor is a function $F(\mathbf{Y}) : \mathbf{y} \rightarrow [0, 1]$ where $\mathbf{y}$ is an instantiation of $\mathbf{Y}$. The probability distribution defined by the graphical model is $\Pr(\mathbf{X}) = \frac{1}{\mathcal{Z}} \prod_{F_i \in \mathbf{F}} F_i(\mathbf{X}_i)$ where $\mathbf{X}_i \subseteq \mathbf{X}$, and $\mathcal{Z}$, termed *partition function*, normalizes the probability to sum to one.

One of the prominent methods for performing exact probabilistic inference in graphical models is the Junction Tree algorithm [18, 13]. Let $G(\mathbf{X}, E)$ be a PGM. A *Junction Tree* for G is a tree $T(\mathtt{C})$, defined over a set of nodes $\mathtt{C}$ that satisfy the following properties:

1. Each node $C_i \in \mathtt{C}$ is associated with a set of variables $\mathbf{Y}_i \subseteq \mathbf{X}$ from the PGM and a factor $G_i(\mathbf{Y}_i)$ (not to be confused with the PGM factors denoted F_i).
2. For each factor $F_k(\mathbf{X}_k)$ in the PGM, there exists a tree node $C_i \in \mathtt{C}$ such that $\mathbf{X}_k \subseteq \mathbf{Y}_i$.
3. If nodes $C_i, C_j \in \mathtt{C}$ are both associated with a variable $X \in \mathbf{X}$, then every node on the path connecting them in T is also associated with X.

The edges of the junction tree are labeled with the intersection of their endpoints. A separator, $\mathbf{S}_{i,j}$, connects nodes C_i and C_j and is referred to as a *separator node*.

Inference in junction trees is performed by passing messages between adjacent clique nodes. Evidence, $\mathbf{E} = \mathbf{e}$ is materialized by eliminating inconsistent factor entries. The message passing is carried out in two phases, inward - from the leaves towards the root, and outward - from the root towards the leaves. A node C_i sends a message to its neighbor, C_j, only after it has received messages from the rest of its neighbors $Nbr_i \setminus C_j$. The message $\mu_{i \rightarrow j}(\mathbf{S}_{ij})$ from node C_i to C_j is a tabular factor defined over their intersection, $\mathbf{S}_{ij} = \mathbf{Y}_i \cap \mathbf{Y}_j$, as follows: $\mu_{i \rightarrow j}(\mathbf{S}_{ij}) = \sum_{\mathbf{Y}_i \setminus \mathbf{S}_{ij}} G_i \prod_{k \in Nbr_i \setminus C_j} \mu_{k \rightarrow i}$. Once message propagation completes, each tree-node factor holds the marginal distribution *i.e.*, $G_i(\mathbf{Y}_i) = \Pr(\mathbf{Y}_i, \mathbf{e})$.

The *width* of a junction tree is the size of its largest node minus one. The treewidth $tw(G)$ of a graph G is the minimum width among all possible junction trees for G. In general, minimizing the graph width is known to be NP-complete. Since the junction-tree algorithm relies on tabular factors for performing the marginalization operation required for message-passing, the runtime of the algorithm depends, exponentially, on its width. Therefore, bounded width implies tractability in graphical models.

3 CNF-trees: Junction Trees for CNFs

In this section we introduce common notation and define CNF-trees, which are specialized junction-trees for Boolean formulas in CNF. The proposed algorithm, described in Section 4.1, operates over this structure.

A *literal* l of a binary variable X is either a variable or its negation, which are denoted by $x, \bar{x}$, respectively. The variable corresponding to a literal l is denoted by $\mathrm{var}(l)$. Each literal, l, is associated with a weight $\mathrm{p}_l \in [0, 1]$. An assignment is a function $\gamma : \mathbf{V} \rightarrow \{0, 1\}$ and will be denoted by its set literals $\gamma = \{l_1, l_2, ..., l_k\}$. An assignment's weight is defined as the product of its literal weights. The *projection* of an assignment γ over a subset of its vars $\mathbf{Y} \subseteq \mathrm{var}(\gamma)$ is denoted $\gamma|\mathbf{Y}$. For example, given the assignment $\gamma = \{x_1, \bar{x}_2, x_3\}$, then $\gamma|\{X_2, X_3\} = \{\bar{x}_2, x_3\}$.

A Boolean formula f over variables $\mathbf{X}$ maps each instantiation $\mathbf{x}$ to either *true* or *false*. $f(\mathbf{X})$ is in Conjunctive Normal Form (CNF), constructed from a conjunction of *clauses*, each a disjunction of literals. We denote by $\phi_1, \phi_2, .., \phi_n$ the set of unique clauses in f, where every ϕ_i represents a set of literals. The variables in a clause ϕ_i are denoted $\mathrm{var}(\phi_i)$, and the clauses of f that contain a literal l are denoted $\mathrm{clauses}(l)$. We assume that the formula f is simplified, meaning, for every pair of clauses $\phi_i, \phi_j \in f$, $\phi_i \nsubseteq \phi_j$. Conditioning a CNF formula f on literal l, denoted $f|l$, consists of removing the literal $\bar{l}$ from all clauses, and dropping the clauses that contain l. Conditioning a formula on an assignment, or a set of literals $\gamma = \{l_1, l_2, ..., l_k\}$, denoted $f|\gamma$, amounts to conditioning it on every literal $l \in \gamma$. We say that an assignment γ is *consistent* if $f|\gamma \neq 0$. We say that a variable X *affects* the formula's outcome if $f|x \neq f|\bar{x}$. We denote by $\mathrm{var}(f)$ the set of variables that affect the formula. A pair of formulae f_1, f_2 are disjoint if $\mathrm{var}(f_1) \cap \mathrm{var}(f_2) = \emptyset$. The *weighted model count* or probability that f is satisfied is denoted by $\Pr(f)$ and the two terms may be used interchangeably.

Let $G_f(\mathbf{X}, E)$ denote the primal graph of $f(\mathbf{X})$, where nodes represent variables and there is an edge between pairs of variables that belong to a common clause.

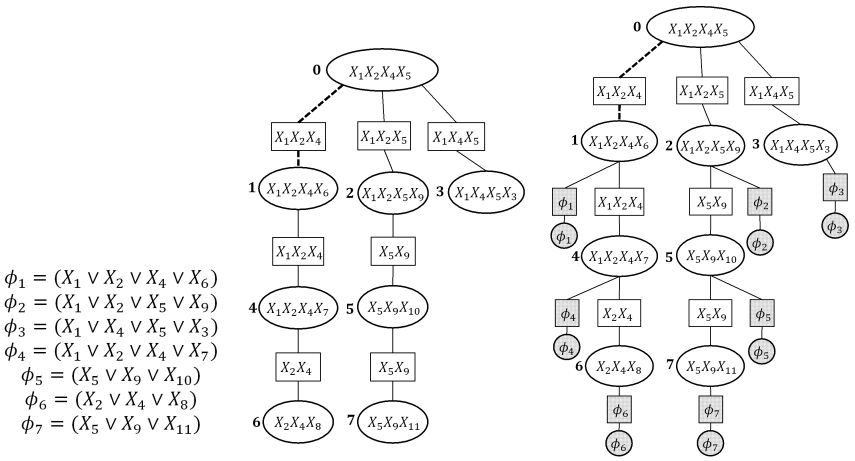
Definition 1 (CNF-tree)**.** *Let $f(\boldsymbol{X})$ be a Boolean formula in CNF with primal graph $G_f(\boldsymbol{X}, E)$. A* CNF-tree *for f is a rooted junction tree, $T_r(\mathtt{C})$, for $G_f(\boldsymbol{X}, E)$ where each clause $\phi_i \in f$ is represented as a leaf node with factor $F_{\phi_i}(\boldsymbol{y})$, $\boldsymbol{Y} \subseteq \mathrm{var}(\phi_i)$:*

$$F_{\phi_i}(\boldsymbol{y}) = \begin{cases} 0 & \text{if } \phi_i|\boldsymbol{y} = 0 \\ 1 & \text{if } \phi_i|\boldsymbol{y} = 1 \\ 1 - \prod_{l \in \phi_i|\boldsymbol{y}} \Pr(\bar{l}) & \text{otherwise} \end{cases} \tag{1}$$

According to the junction tree properties, each clause, $\phi_i \in f$, is associated with a node $C_i \in \mathtt{C}$ such that $\mathrm{var}(\phi_i) \subseteq \mathbf{X}_i$. This node-clause relationship is reflected in the tree by attaching a leaf, representing the clause, to its associated tree-node. For example, consider the CNF formula, its junction and CNF-trees in Fig. 1. The shaded leaf nodes represent clauses.

Configuring the leaf-node factors to return the probability that their respective clause is satisfied is equivalent to introducing evidence which prohibits assignments that falsify the formula. Thereby, the weighted model count of f can be performed by message-passing on the CNF-tree.

In the general setting, a separator set $\mathbf{S}_{i,j} = \mathbf{X}_i \cap \mathbf{X}_j$, between junction tree nodes C_i, C_j, enables inducing independence between variables on different sides of the edge only when *all* of the variables in $\mathbf{S}_{i,j}$ were assigned a value [8]. We observe, however, that in CNF-trees this requirement may be too strict as illustrated in Example 1.



(a) f (b) Junction tree for f (c) CNF-tree for f

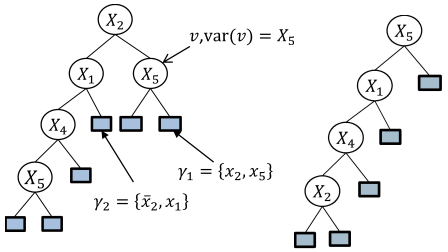
Fig. 1: A formula, $f = \wedge_{i=1}^{7}\phi_i$, and its corresponding junction and CNF trees

Example 1. Consider the CNF-tree in Fig. 1. The *partial* assignment $\gamma_1 = \{x_1\}$ renders the disjoint variable-sets on the two sides of the edge, (C_0, C_1) (marked) independent, even though variables X_2 and X_4 remain unassigned. The reason for this is that given x_1, the original formula is reduced to $f|x_1 = \underbrace{\phi_6}_{f_1}\underbrace{\phi_5\phi_7}_{f_2}$. Variables X_3, X_6, and X_7 become irrelevant to f's outcome following the partial assignment x_1, and can be disregarded. The variables that belong to the disjoint components in the reduced formula $f|x_1$, namely $\mathrm{var}(f_1)$ and $\mathrm{var}(f_2)$, are conditionally independent given x_1.

Example 1 motivates the search of a set of partial assignments to the factors of a CNF-tree that will render their subtrees independent. Partial, rather than complete assignments, may reduce factor sizes, enabling more efficient inference.

4 Decomposition by Partial Assignments

The next two sections lay out the main contribution of the paper. We first explain how the CNF-tree structure can be utilized for generating tree-Conditional-Probability-Tables (tree-CPTs) [2] consisting of a small set of mutual exclusive and exhaustive partial assignments, which are guaranteed to decompose a formula. We then define tree-CPT cardinality and suggest optimizations for size reduction. Tree-CPTs, introduced in [2], is a representation which captures Context-Specific-Independence which



(a) F_0 (b) F_0'

Fig. 2: Two possible tree-CPTs for root-node C_0 ($\mathbf{X}_0 = \{X_1, X_2, X_4, X_5\}$) of the junction tree in Fig. 1c.

can be exploited for probabilistic inference. We adopt this structure in order to represent partial assignments of CNF-tree-node members, but give it a different semantics.

Let $T_r(\mathtt{C})$ be a CNF-tree rooted at node r. For each tree node $C_i \in \mathtt{C}$, $F_i(\mathbf{X}_i)$ is the factor associated with this node, T_i is the subtree rooted at C_i, and f_i is the subformula induced by the clauses in this subtree. For example, the formulae represented by subtrees T_1, T_2 rooted at nodes C_1, C_2, respectively, in Fig. 1c are $f_1 = \phi_1\phi_4\phi_6$, and $f_2 = \phi_2\phi_5\phi_7$. The children of tree-node C_i are denoted ch_i.

Definition 2. *Let $T_r(\mathtt{C})$ be a CNF-tree rooted at node r and $C_i \in \mathtt{C}$ a node in T_r with $\mathrm{ch}_i = \{C_1, C_2, ..., C_m\}$. Let $\boldsymbol{Y} = \boldsymbol{y}$ be a partial assignment to $\boldsymbol{X}_i$. $\boldsymbol{y}$ is called a* valid (partial) assignment *to $\boldsymbol{X}_i$ if the following two conditions are satisfied:*

1. *$\boldsymbol{Y} = \boldsymbol{y}$ is consistent ($f_i|\boldsymbol{y} \neq 0$)*
2. *$f_i|\boldsymbol{y}$ is decomposed to sub-formulas $f_1|\boldsymbol{y}, f_2|\boldsymbol{y}, ..., f_m|\boldsymbol{y}$, which are pairwise disjoint.*

Our goal is to generate the smallest factor for each CNF-tree node. Namely, per each node, we would like to identify the smallest set of mutual exclusive and exhaustive *valid* partial assignments (Def. 2). The factors will be represented by tree-CPTs [2] where non-terminal vertices represent variables and terminal vertices correspond to the assignment defined by the path from the root. The variable corresponding to a vertex v in the tree-CPT is denoted $\mathrm{var}(v)$, its parent $p(v)$, and its right and left children corresponding to assignment $\mathrm{var}(v) = 1/0$ as v^r/v^l, respectively. The set of assignments represented by terminal nodes in F_i will be denoted $\boldsymbol{\gamma_i}$, their cardinality $k_i = |\boldsymbol{\gamma_i}|$, and for each $\gamma \in \boldsymbol{\gamma_i}$, the assignment's marginal probability will be denoted $\Pr(\gamma)$. For each vertex v in the tree-CPT, we denote the path from the root to v, and the assignment it dictates, by P_v. The assignment P_v will be referred to as v's *context*. We say that literal $l \in P_v$ if $l|P_v = 1$.

Example 2. An example of two possible tree-CPTs for root-node C_0 in the CNF-tree of Fig. 1c appear in Fig. 2. Note the terminal vertex in Fig. 2a that represents assignment $\gamma_2 = \{\bar{x}_2, x_1\}$. The partial assignment γ_2 induces subformulae $f_1|\gamma_2 = \phi_6$, $f_2|\gamma_2 = \phi_5\phi_7$, and $f_3|\gamma_2 = \emptyset$, which correspond to subtrees T_1, T_2, T_3, respectively. Given assignment γ_2, these subformulae are consistent and pairwise disjoint, *e.g.*, $\mathrm{var}(f_1|\gamma_2) \cap \mathrm{var}(f_2|\gamma_2) = \emptyset$, thus the partial assignment γ_2 is valid.

Algorithm 1: $\texttt{MP}(T_r, \boldsymbol{e})$, **returns** $\Pr(f_r|\boldsymbol{e})$

```
1  if r is a leaf-node then
2      return F_r(e) // By eq. 1
3  if cache(f_r|e) ≠ nil then
4      return cache(f_r|e)
5  Pr(f_r|e) ← 0.0 // init the return value
6  while γ ← nextValid(γ, f_r|e, X_i \ var(e)) ≠ nil do
7      Pr(f_r|eγ) ← ∏_{l∈γ} p_l // assignment weight
8      foreach node n ∈ ch_r do
9          Pr(f_n|eγ) ← MP(T_n, eγ) // recurse
10         Pr(f_r|eγ) ← Pr(f_r|eγ) · Pr(f_n|eγ)// Thm. 1
11     Pr(f_r|e) ← Pr(f_r|e) + Pr(f_r|eγ) // Thm. 1
12 cache(f_r|e) ← Pr(f_r|e)
13 return Pr(f_r|e)
```

4.1 Message-Passing in CNF-trees

The procedure for performing WMC over CNF-trees is presented in Alg. 1, taking a CNF-tree T_r, which represents f_r, and a partial (possibly empty) assignment $\boldsymbol{e}$, and returning the probability that $f_r|\boldsymbol{e}$ is satisfied. The algorithm avoids repeated computation of equivalent CNFs using a cache whose key represents the CNF. The function `nextValid` (Line 6) retrieves the next valid assignment to process. We detail the generation of valid assignments in Section 5.

Thm. 1 establishes the soundness of the algorithm. Its proof is inductive and follows from the validity (Def. 2) of the tree-CPT assignments. Due to space constraints proofs are omitted.

Theorem 1. *Let T_r be a CNF-tree representing CNF f_r. The call $\texttt{MP}(T_r, \boldsymbol{e})$ returns $\Pr(f_r|\boldsymbol{e})$.*

5 Generating small tree-CPTs

Alg. 1 motivates the search for small tree-CPTs. Each tree-CPT internal vertex induces an instantiation of the variable it represents. Therefore, we begin by observing the conditions that forgo the requirement to instantiate a variable.

Definition 3 (safe variable)**.** *Let C_i be a CNF-tree node with arguments $\boldsymbol{X}_i$, and let γ be an assignment. A variable $X \in \boldsymbol{X}_i$ is called* safe *if there is at most a single node, $C_j \in \text{ch}_i$, such that $X \in \text{var}(f_j|\gamma)$. The set of variables in $\boldsymbol{X}_i$ that are safe under assignment γ are denoted* $\text{safe}_i(\gamma)$.

We first note that by Def. 3, instantiated variables are safe because they cannot appear in any induced sub-formula associated with a node's subtrees.

To relate safe variables to compact CPT-trees let $F_i(\mathbf{X}_i)$ be C_i's CPT-tree, and let vertex $v \in F_i$ have context P_v. If $X \in \text{safe}_i(P_v)$, and P_v is consistent (recall, $f_i|P_v \neq$

0) then, by Def. 3, there is a valid assignment that contains P_v, but not X. Furthermore, a context P_v for which all the node arguments are safe, is a valid assignment by definition.

The variation between tree-CPTs, and hence the efficiency of the WMC algorithm, stems from the different ordering of variable instantiation (see Figure 2). Def. 4 gives the ordering constraints between the arguments of a node C_i, which will be used to derive its tree-CPT.

Definition 4 (Conditioning graph)**.** *The* conditioning graph *of a CNF-tree-node C_i is a directed graph $D_i(L_i, E_i)$, where $L_i = \{x, \bar{x} : X \in \boldsymbol{X}_i\}$ is the set of literals of $\boldsymbol{X}_i$. There is an edge $(l_1, l_2) \in E_i$ if $\exists \phi_1, \phi_2 \in f_i$ such that*

1. $\phi_1, \phi_2 \in \text{clauses}_i(\text{var}(l_1)) \setminus \text{clauses}_i(l_2)$
2. *Node C_i is their Least Common Ancestor (LCA) in the CNF-tree.*

The compliment of D_i is denoted $\bar{D}_i$.

The intuition behind the conditioning graph becomes apparent when looking at absent edges, or at the conditioning-graph's compliment, $\bar{D}_i$. If, for example, $x_1 \rightarrow x_2 \in \bar{D}_i$, that is, $x_1 \rightarrow x_2 \notin D_i$, then, by Def. 4, any two clauses that contain X_1 (*i.e.*, x_1 or $\bar{x}_1$), but not the literal x_2, are confined to the same subtree of node C_i. Practically, this means that given an assignment in which $x_2 = 1$, the set of unsatisfied clauses containing variable X_1 are confined to (at most) a single subformula represented by one of C_i's subtrees. In other words, variable X_1 is *safe* (Def. 3) for any assignment where x_2 is set. Essentially, given two literals, l_1 and l_2, the conditioning graph answers the following question: "Given $l_2 = 1$ is $\text{var}(l_1)$ safe ?". If $l_1 \rightarrow l_2 \in \bar{D}_i$ then the answer is affirmative, otherwise negative.

We also note the following about the conditioning graph and its compliment. First, for each variable $X \in \mathbf{X}_i$, $\text{out}_i(x) = \text{out}_i(\bar{x})$ because the out-edges are determined by the existence of a variable (*i.e.*, literal x or $\bar{x}$) in the clauses of Def. 4. Also, since assigning a variable makes it safe, then neither the conditioning graph nor its compliment contain edges from a literal to its compliment or self-loops.

Example 3. Fig. 3 presents the conditioning graph of the root C_0 of the CNF-tree in Fig. 1c. The edge $x_5 \rightarrow x_2$ ($\bar{x}_5 \rightarrow x_2$) is due to clauses ϕ_7 and ϕ_3. Both clauses contain x_5 but not x_2, and their least common ancestor in the CNF-tree is C_0.

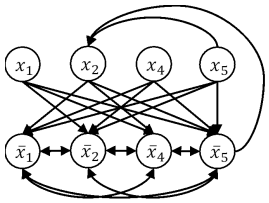


Fig. 3: The Conditioning Graph, D_0 of node C_0 of the CNF-tree in Fig. 1c

We now characterize a subclass of safe variables in context P_v (Def. 3), denoted $\mathbf{Y}_v$, using the conditioning graph.

Definition 5. *Let C_i denote a node in a rooted CNF-tree with conditioning graph D_i, and tree-CPT $F_i(\boldsymbol{X}_i)$. Let v be a vertex in the tree-CPT F_i, with context P_v, then:*

$$\boldsymbol{Y}_v = \{X \in \boldsymbol{X}_i : \exists l \in P_v : x \rightarrow l \in \bar{D}_i\}$$

Theorem 2. *Let v be a vertex in F_i with context P_v. Then $\mathbf{Y}_v \subseteq \mathrm{safe}_i(v)$.*

Thm. 2 characterizes the set of arguments whose clauses are confined to a single subformula induced by the subtrees of node C_i, given the assignment dictated by P_v. By Thm. 2, X will not require instantiation in order to extend P_v to a valid assignment. The importance of the variable-set $\mathbf{Y}_v$ stems from the fact that it can be *statically* identified by considering only the structure of the CNF-tree.

The complement of the variable set $\mathbf{Y}_v$, described in Def. 5, has incoming edges to all of the literals set by the assignment P_v. We define this set of variables, $\mathbf{Z}_v$, recursively as follows.

Definition 6. *Let v be a vertex in CPT-tree F_i, with parent $p(v)$, and variable $\mathrm{var}(p(v))$. The variable-set $\mathbf{Z}_v$ is:*

$$\mathbf{Z}_v = \begin{cases} \boldsymbol{X}_i & \textit{if } P_v = \emptyset \textit{ or } v \textit{ is the root} \\ \mathbf{Z}_{p(v)} \cap \mathrm{in}_i(\mathrm{var}(p(v))) & \textit{if } \mathrm{var}(p(v)) = 1 \\ \mathbf{Z}_{p(v)} \cap \mathrm{in}_i(\overline{\mathrm{var}(p(v))}) & \textit{if } \mathrm{var}(p(v)) = 0 \end{cases}$$

According to Def. 5 and 6, we have that $\mathbf{Y}_v \cap \mathbf{Z}_v = \emptyset$, and for every vertex $v \in F_i$, $\mathbf{X}_i = \mathbf{Y}_v \cup \mathbf{Z}_v \cup \mathrm{var}(P_v)$.

Example 4. Let v refer to the right child of X_2 in the tree-CPT of Fig. 2a ($\mathrm{var}(v) = X_5$). Then $\mathbf{Z}_v = \mathbf{Z}_{p(v)} \cap \mathrm{in}_0(x_2) = \{x_1, x_2, x_4, x_5\} \cap \{x_5\} = \{x_5\}$. In this case variable X_5 requires instantiation in context x_2 in order to extend the partial assignment $\{x_2\}$ to one that is valid.

5.1 Tree-CPT cardinality

To express the size of a node's tree-CPT $F_i(\mathbf{X}_i)$ with children ch_i and conditioning graph D_i we denote (with a slight abuse of notation) the variable associated with each tree-CPT vertex v, $\mathrm{var}(v) = V$, and its literals v and $\bar{v}$ respectively. $T : \mathbf{Z}_v \rightarrow \mathbb{N}$ maps $\mathbf{Z}_v$ to the number of valid assignments in the subtree rooted at vertex v:

$$T(\mathbf{Z}_v) = \begin{cases} 1 & \text{if } \mathbf{Z}_v = \emptyset \\ T\left(\mathbf{Z}_v \cap \mathrm{in}_i(v)\right) + T\left(\mathbf{Z}_v \cap \mathrm{in}_i(\bar{v})\right) & \text{o.w} \end{cases} \tag{2}$$

This expression considers only the variable set $\mathbf{Z}_v$ because by Thm. 2, the members of the compliment set, $\mathbf{Y}_v$, are safe (Def. 3), and thus do not require instantiation.

When $\mathbf{Z}_v = \emptyset$ then no variable requires instantiation, and a single terminal node can represent the valid assignment. Otherwise, the total size of the tree-CPT rooted at vertex v is determined by the size of the tree-CPTs rooted at its left and right children v^l, v^r respectively. By Def. 6, $\mathbf{Z}_{v^l} = \mathbf{Z}_v \cap \mathrm{in}_i(\bar{v})$ and $\mathbf{Z}_{v^r} = \mathbf{Z}_v \cap \mathrm{in}_i(v)$. It is easy to

see that repeated expansion of Eq. 2 can lead to the known exponential bound for the number of valid assignments in the tree-CPT, whenever for each tree-CPT vertex v:

$$\mathbf{Z}_v \cap \text{in}_i(v) = \mathbf{Z}_v \cap \text{in}_i(\bar{v}) = \mathbf{X}_i \setminus \{\text{var}(P_v) \cup \{V\}\}$$

That is, in the worst case the complexity of Algorithm 1 is exponential in the size of the largest node in the CNF-tree, or the width of the formula's primal graph. In Section 6 we show that despite this worse-case behavior, and with the assistance of the optimization discussed next, Algorithm 1 performs well on known benchmarks.

5.2 Optimizations for generating small tree-CPTs

We broadly address two types of optimizations that we apply to Alg. 1, aimed at minimizing the size of the tree-CPTs. The first is a heuristic that selects the next node-member to assign, using the analysis in Sec. 5.1. The second is Unit Propagation and conflict directed clause learning, adapted from the CNF-SAT domain.

A tree-CPT can be viewed as a binary decision tree where terminal nodes identify valid assignments. Constructing an optimal decision tree, one with fewest nodes, is generally an NP-hard problem [16]. We apply a heuristic strategy that, at each stage, selects the variable that minimizes the cardinality of the variable-set that is common to its left and right tree-CPTs. Formally:

$$\arg\min_v \left(|\mathbf{Z}_v \cap \text{in}_i(v)) \cap (\mathbf{Z}_v \cap \text{in}_i(\bar{v})|\right)$$

Ties may be broken by selecting the variable that further minimizes the set of unsafe variables at either of its sub trees. That is:

$$\arg\min_v \left[\max\left(|\text{in}_i(v) \cap \mathbf{Z}_v|, |\text{in}_i(\bar{v}) \cap \mathbf{Z}_v|\right)\right]$$

Unit Propagation (UP) refers to the process of iteratively assigning literals of unit clauses until none are left. It is part of both DPLL-based model counters [23, 24] and compilers that generate d-DNNF circuits [6, 21]. Specifically, if $\phi = \{l\}$ is a unit clause of a CNF formula f, then the UP process deletes all occurrences of $\bar{l}$, and all clauses containing l, which are now satisfied. Each valid assignment generated by Alg. 1 is extended by applying unit propagation. That is, the valid assignments are guaranteed to decompose the formula and ensure that no unit clauses are present. Unit propagation is applied after every variable assignment during the tree-CPT construction.

If UP results in a conflict, then a new clause is learned by applying the first Unique Implication Point schema [20]. The newly learned clause is added to the subformula being processed, f_i. We note that the learned clauses are used only during unit propagation, in order to detect conflicts early. They are not represented as leaves in the CNF-tree, and are not considered during caching. Also, since different nodes represent different subformulas, then each CNF-tree node holds its own local set of conflict clauses.

6 Empirical Evaluation

We evaluate the proposed approach on a set of benchmark networks from the UAI probabilistic inference challenge.[1] We compare our results with the C2D compiler [6], part of the Ace system.[2] Besides evaluating the efficiency of the proposed approach, we discuss the properties of networks that benefit from it. The experimental setup is given in Section 6.1, followed by results and analysis in Section 6.2. We implemented our algorithm in C++ [3] and carried out the experiments on a 2.33GHz quad-core AMD64 with 8GB of RAM running CentOS Linux 6.6. Individual runs were limited to a 2000-second time-out.

6.1 Overview and Methodology

The compilation process of C2D is guided by a binary tree, termed *dtree* whose leaves are associated with the clauses of f. The dtree determines the instantiation order materialized in the d-DNNF [5, 6]. Fig. 4 depicts a dtree of the CNF of Fig. 1a. Each internal node, T, is associated with a variable-set, called *separator* [12], which is the variable-set common to the left and right subtrees of the node. Once these variables have been assigned, the formulae represented by the two subtrees become disjoint. Darwiche [6] observed that there is no need to set all variables in the dtree-node T in order to decompose the formula. That is, after setting a subset of the dtree-node variables, enough clauses may become satisfied such that the rest of the T's variables are no longer shared between the formulas represented by its left and right children. For this reason the C2D compiler *recomputes* the separator for T each time a variable of T is decided [6]. Within each separator, the C2D compiler chooses the variable that appears in the largest number of unsatisfied clauses.

Darwiche shows that the clusters of a dtree satisfy the junction-tree property ([8], Thm. 9.10). That is, the maximal clusters of a dtree can be connected such that they constitute a junction-tree. Once the junction-tree is created, we can attach the clauses as leaf nodes to obtain the CNF-tree. Applying our algorithm to a junction tree corresponding to the dtree generated by the C2D compiler, enables comparing the two approaches on an even ground, although our proposed approach is not limited to binary junction trees. Furthermore, we can gain insight into the types of networks that benefit from our proposed approach, which requires more analysis at each junction tree node.

There is a wide range of settings for C2D, and in particular for generating the dtree. We experimented with the default provided by Ace, termed dtBnMinfill. This option instructs the program to generate a dtree for the original Bayesian network using the *minfill* heuristic [17], which is widely known for generating small induced width elimination orders. Each leaf in the resulting dtree corresponds to one of the network CPTs. Then, each leaf is replaced with the dtree that represents the corresponding CPT.

6.2 Experimental results

We report the results obtained for Grid, Promedas, and Segmentation networks. The evaluation is presented using scatter plots. Each instance is represented as a point in

[1] Available online at http://www.cs.huji.ac.il/project/PASCAL/showNet.php
[2] Available online at http://reasoning.cs.ucla.edu/ace/
[3] Code is available at: https://github.com/batyak/PROSaiCO/

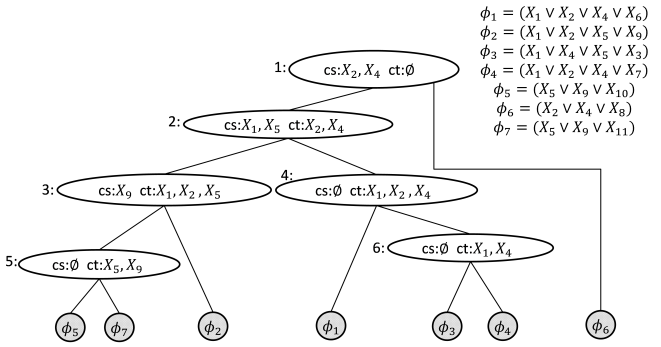


Fig. 4: CNF and corresponding dtree.

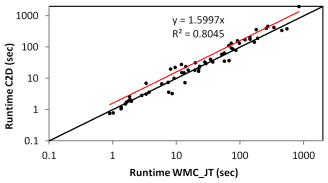
the chart whose x, y coordinates represent runtime, in seconds, of WMC_JT and C2D, respectively. Points above the $y = x$ line represent problem instances where WMC_JT performs better. Axes are log-scale. We also mark a linear trendline (which translates to exponential trendline due to the log-scale) with it R^2 value.

Grid networks The nodes in random grid networks represent binary variables that are arranged in an $N \times N$ square. Each CPT is generated uniformly at random. The fraction of the nodes that are assigned deterministic CPTs, having only 0 and 1 probability entries, is captured by the *deterministic ratio*.

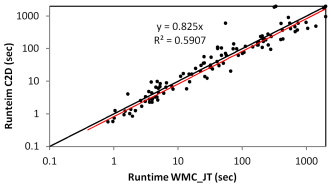
Figure 5 shows the results over the grid networks for three deterministic ratios. All networks solved by at least one of the solvers are present. On the grids with a 50% deterministic ratio, WMC_JT outperformed C2D on 39 out of the 60 instances. On one instance, C2D did not complete within the designated timeout. On the grids with a 75% deterministic ratio, WMC_JT outperformed C2D on 43 out of the 110 instances, while C2D outperformed WMC_JT on 63 instances. On the 90% benchmark, WMC_JT outperformed C2D on only 13 of the 100 instances. Therefore, we can conclude that in general, with less determinism WMC_JT tends to outperform C2D. When the underlying network contains a large percentage of deterministic factors (90%), a small fraction of the node members determine the rest through UP and the time invested by WMC_JT in the generation of conditioning graphs and small tree-CPTs may be too costly.

Promedas and Segmentation Promedas stands for " PRObabilistic MEdical Diagnostic Advisory System". The Promedas benchmark contains 238 Markov networks, consisting of binary variables, which were converted from layered noisy-or Bayesian networks that represent real-world medical diagnosis cases. The networks' treewidth is up to 60, and many of them are considered too difficult for exact algorithms[4]. Results are plotted in Fig. 6a. Out of the 238 networks, WMC_JT processed 102 networks within the designated timeout, while C2D completed 89. Out of the 89 networks processed by both algorithms, C2D outperformed WMC_JT on 65, while WMC_JT outperformed C2D on 24. On the Segmentation benchmark, Fig. 6b, WMC_JT outperformed C2D on all 50 instances.

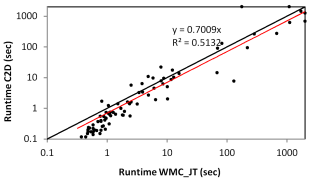
[4] http://graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks



(a) deterministic ratio: 50%, $12 \leq N \leq 18$,



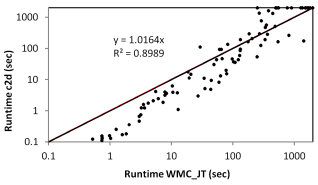
(b) deterministic ratio: 75%,$16 \leq N \leq 26$



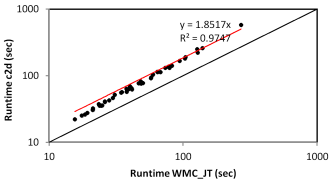
(c) deterministic ratio: 90%, $20 \leq N \leq 38$

Fig. 5: Grid networks: Points above the $y = x$ line represent instances where `WMC_JT` is better.

Overall, we observe that `WMC_JT` tends to outperform `C2D` over instances with a low to medium percentage of deterministic factors. Furthermore, the results and trendlines of the Promedas and Segmentation benchmarks (Fig. 6) suggest that the relative performance of `WMC_JT` improves on "harder" instances, those which require more CPU cycles for both algorithms. That said, we note that the algorithm execution time is determined by many variables. These include the d-tree used, its orientation and the variable order generated by the heuristic described in Sec. 5.2.



(a) Promedas



(b) Segmentation

Fig. 6: Points above the $y = x$ line represent instances where `WMC_JT` is better.

7 Related Work

We position our work along two dimensions: *exhaustive search* vs. *knowledge compilation*, and *dynamic* vs. *static decomposition*.

DPLL-based algorithms exhaustively explore the search-tree for a formula, while pruning unsatisfiable branches. At the heart of the search-based techniques for weighted model counting are two operations, a Shannon expansion on a decision variable Z, that

is, $\Pr(f) = \Pr(f|\bar{z})\Pr(\bar{z}) + \Pr(f|z)\Pr(z)$ and the partitioning of the formula into disjoint components [12]. Extensions that tremendously improve the performance of DPLL-based algorithms include non-chronological backtracking, [1], conflict directed clause-learning (CDCL), and variable branching heuristics [23].

In *knowledge compilation*, the formula is compiled into a representation that enables computing the probability of evidence in time that is polynomial in its size [4, 21]. These representations are based on Negation Normal Form (NNF) circuits [7] where internal nodes represent either conjunctions or disjunctions and leaf nodes represent constants or literals. Circuits that enable tractable model counting, termed deterministic-DNNF (d-DNNF), must be decomposable and deterministic. The former requires children of conjunction nodes to share no variables, and the latter requires children of disjunction nodes to be mutual exclusive. State-of-the-art model counting compilers, `C2D` [6] and `DSharp` [21], generate Decision-DNNF circuits that ensure determinism as follows. Each *or* node, n, is associated with a variable X such that n's right and left children represent subformulas $f_n|x$, and $f_n|\bar{x}$ respectively. This method of ensuring determinism is closely related to the instantiation step of DPLL-based algorithms [14, 15]. Our proposed approach fits knowledge compilation, where the CNF-tree may be reused to answer different queries.

Static variable instantiation order is used to compile formulas to Ordered Binary Decision Diagrams (OBDDs) [3]. In contrast, a fully dynamic order, applied in DPLL-based algorithms, becomes effective in formulae that can be decomposed by a small number of well selected variables. DPLL-based algorithms attempt to decompose the formula into disjoint components after each instantiation. Nevertheless, despite the use of clever heuristics [22], there is no guarantee to the effectiveness of the instantiation in terms of partitioning the residual formula into disjoint components [12], making fully-dynamic variable instantiation inefficient when applied to heavily connected formulae.

The approach presented in this paper, as well as the dtree-guided `C2D` approach, may be considered semi-dynamic because the variable instantiation order is largely determined by the structure and orientation of the CNF-tree (or dtree). Our approach, however, takes a more holistic view and identifies the set of valid *assignments* that are guaranteed to decompose the formula. It also makes a deliberate effort to minimize the cardinality of this set by careful ordering of the node members.

8 Conclusions

We present CNF-trees of Boolean formulae to reveal structure that can be used to enhance the performance of WMC algorithms. We present a method for utilizing this structure in order to generate small tree-CPTs, and evaluate it over a set of known benchmarks. As part of future research we intend to characterize CNF-trees that enable efficient WMC.

Acknowledgments

The work was carried out in and partially supported by the Technion–Microsoft Electronic Commerce research center.

References

1. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *FOCS*, pages 340–351, 2003.
2. C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in bayesian networks. In *UAI*, pages 115–123, 1996.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.
4. M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, Apr. 2008.
5. A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48:2001, 2001.
6. A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *ECAI*, pages 328–332, 2004.
7. A. Darwiche and P. Marquis. A knowledge compilation map, 2002.
8. P. A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
9. R. Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.
10. C. Domshlak and J. Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *CoRR*, abs/1111.0044, 2011.
11. C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. IOS Press, 2009.
12. C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225 – 263, 1996.
13. J. Huang and A. Darwiche. Dpll with a trace: From sat to knowledge compilation. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 156–162. Professional Book Center, 2005.
14. J. Huang and A. Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29:191–219, 2007.
15. L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15 – 17, 1976.
16. U. Kjrulff. Triangulation of graphs – algorithms giving small total state space. Technical report, 1990.
17. S. L. Lauritzen and D. J. Spiegelhalter. Readings in uncertain reasoning. chapter Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems, pages 415–448. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
18. J. P. Marques Silva and K. A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pages 467–469. IEEE, 1996.
19. C. J. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: Fast d-dnnf compilation with sharpsat. In *Proceedings of the Canadian Conference on Artificial Intelligence*, pages 356–361, 2012.
20. T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *SAT*, pages 226–240, 2005.
21. T. Sang, P. Beame, and H. A. Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, pages 475–482, 2005.
22. M. Thurley. sharpsat - counting models with advanced component caching and implicit bcp. In *SAT*, pages 424–429, 2006.
23. L. G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.