Generating Random Weighted Model Counting Instances: An Empirical Analysis with Varying Primal Treewidth

Abstract

Weighted model counting (WMC) is an extension of propositional model counting with applications to probabilistic inference and other areas of artificial intelligence. In recent experiments, WMC algorithms are shown to perform similarly overall but with significant differences on specific subsets of benchmarks. A good understanding of the differences in the performance of algorithms requires identifying key characteristics that favour some algorithms over others. In this paper, we introduce a random model for WMC instances with a parameter that influences primal treewidth—the parameter most commonly used to characterise the difficulty of an instance. We then use this model to experimentally compare the performance of WMC algorithms C2D, CACHET, D4, DPMC, and MINIC2D on random instances. We show that the easy-hard-easy pattern is different for algorithms based on dynamic programming and algebraic decision diagrams (ADDs) than for all other solvers. We also show how all WMC algorithms scale exponentially with respect to primal treewidth and how this scalability varies across algorithms and densities. Finally, we demonstrate how the performance of ADD-based algorithms changes depending on how much determinism or redundancy there is in the numerical values of weights.

1 Introduction

Weighted model counting (WMC)—a weighted generalisation of propositional model counting (#SAT) (Chavira and Darwiche 2008)—has emerged as a powerful computational framework for problems in a variety of domains. In particular, WMC has been used to perform probabilistic inference for graphical models such as Bayesian networks and Markov random fields (Bart et al. 2016; Chavira and Darwiche 2005, 2006; Darwiche 2002; Sang, Beame, and Kautz 2005b), probabilistic programs (Holtzen, Van den Broeck, and Millstein 2020), and probabilistic logic programs (Fierens et al. 2015). More recently, WMC was used in the context of neural-symbolic artificial intelligence as well (Xu et al. 2018). Extensions of WMC add support for continuous variables (Belle, Passerini, and Van den Broeck 2015), infinite domains (Belle 2017), and first-order logic (Van den Broeck et al. 2011; Gogate and Domingos 2016) and generalise the definition to support arbitrary pseudo-Boolean functions instead of clauses (Dilkas and Belle 2021b). All exact WMC algorithms can be broadly classified as based on search (Sang et al. 2004), knowledge compilation (Darwiche 2004; Lagniez and Marquis 2017; Oztok and Darwiche 2015), or dynamic programming (Dudek, Phan, and Vardi 2020a,b). Other approaches to WMC include approximate (Renkens et al. 2014) and parallel algorithms (Dal, Laarman, and Lucas 2018; Fichte et al. 2018), quantum computing (Riguzzi 2020), and reduction to model counting (Chakraborty et al. 2015).

Recent papers that include experimental comparisons of WMC algorithms show many of them performing very similarly overall (Dudek, Phan, and Vardi 2020a,b) but with overwhelming differences when run on specific subsets of data (Dilkas and Belle 2021a,b; Lagniez and Marquis 2017). Examples of such segregating data sets include bipartite Bayesian networks by Sang, Beame, and Kautz (2005b) and relational Bayesian networks by Chavira, Darwiche, and Jaeger (2006) that encode reachability in graphs under node deletion. So far, such performance differences remain unexplained. However, knowledge about the nature of these differences can inform our choices and aid in further algorithmic developments. Moreover, identifying performance predictors of algorithms is often an important step in developing a portfolio approach to the problem (Xu et al. 2008). Lastly, if new algorithms are always tested on the same set of benchmarks, eventually they may become somewhat fitted to the particular characteristics of those instances, leading to algorithms that may perform worse when run on new types of data (Hossain et al. 2010).

Both theoretical and experimental analysis of SAT (and, to a lesser extent, #SAT) algorithms on random instances is a rich area of research spanning almost forty years. Variations of some of the first random models ever proposed (Franco and Paull 1983; Purdom Jr. and Brown 1983) continue to be instrumental up to this day for, e.g., establishing the location of the threshold between satisfiable and unsatisfiable instances (Achlioptas and Moore 2002) and efficiently approximating #SAT (Galanis et al. 2020). Other random models consider non-uniform variable frequencies (Ansótegui, Bonet, and Levy 2009), fixing the number of times each variable occurs both positively and negatively (Coja-Oghlan and Wormald 2018), and adding other constraints such as cardinality and 'exclusive or' (Pote, Joshi, and Meel 2019). In contrast, only one WMC algorithm so far has been analysed using random instances (Sang et al. 2004; Sang, Beame, and Kautz 2005a). Similarly, while there is a recent attempt (Dilkas and Belle 2020) to compare WMC algorithms on random instances of a particular application of WMC (i.e., probabilistic logic programs), it fails to discern any meaningful differences among the algorithms. The goal of this paper is to explain some of the differences between WMC algorithms via an experimental study that uses random instances.

Experimental work investigating how SAT algorithms behave on random instances is typically centred around parameters that describe each instance independently of its size. The most well-known parameter is the ratio of clauses to variables (i.e., *(clause) density*). Early work in the area showed random 3-SAT instances to be at their hardest when density is around 4.25 (Mitchell, Selman, and Levesque 1992). Later work revealed that the interaction between density and empirical hardness is much more solver-dependent (Coarfa et al. 2003). Many other parameters such as heterogeneity, locality, and modularity have emerged from attempts to generate random instances similar to industry benchmarks for SAT (Ansótegui, Bonet, and Levy 2009; Bläsius, Friedrich, and Sutton 2019; Giráldez-Cru and Levy 2016, 2017).

What parameter(s) are most appropriate to study WMC? Theoretical upper bounds on the performance of various WMC algorithms typically include a factor exponential in the primal treewidth of the input formula (or a closely related notion) (Bacchus, Dalmao, and Pitassi 2009; Darwiche 2001a, 2004; Sang et al. 2004). However—as we show in Section 4—instances generated by a standard random model for k-CNF formulas fail to exhibit enough variance in primal treewidth for us to infer its effect on the behaviour of the algorithms. Therefore, we present an extension of this model with a parameter that influences primal treewidth. The performance of WMC algorithms that use data structures called *algebraic decision diagrams* (ADDs) (Bahar et al. 1997) is also known to depend on the numerical values of weights (Dudek, Phan, and Vardi 2020a,b). Thus, our random model also includes two parameters that control redundancies in these values. We also investigate the effect of redundant weight values (e.g., having weights set to zero and one or having the same weight repeat many times) on the running times of the algorithms.

In addition to introducing a new random model for WMC instances, the contributions of this paper include several key experimental findings about the behaviour of WMC algorithms—namely, C2D[1] (Darwiche 2004), CACHET[2] (Sang et al. 2004), D4[3] (Lagniez and Marquis 2017), DPMC[4] (Dudek, Phan, and Vardi 2020b), and MINIC2D[5] (Oztok and Darwiche 2015)—on random instances. First, we show that the easy-hard-easy pattern with respect to (w.r.t.) density is different for dynamic programming algorithms than it is for all other algorithms. Second, we present statistical evidence that all the algorithms scale exponentially w.r.t. primal treewidth and estimate how the base of that exponential changes w.r.t. density. Third, we show how the performance of ADD-based algorithms gradually improves w.r.t the proportion of weights that have repeating values and sharply improves w.r.t. the proportion of weights set to zero and one.

[1]http://reasoning.cs.ucla.edu/c2d/

[2]https://cs.rochester.edu/u/kautz/Cachet/

[3]https://www.cril.univ-artois.fr/KC/d4.html

[4]https://github.com/vardigroup/dpmc

[5]http://reasoning.cs.ucla.edu/minic2d/

2 Preliminaries

By *variable*, we always mean a Boolean variable. A *literal* is either a variable (say, v) or its negation (denoted $\neg v$), respectively called *positive* and *negative* literal. A *clause* is a disjunction of literals. A *formula* is any well-formed expression consisting of variables, negation, conjunction, and disjunction. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, and it is in k-CNF if every clause has exactly k literals. While we use the set-theoretic notation for CNF formulas (e.g., writing $c \in \phi$ to mean that clause c is one of the clauses in formula ϕ), duplicate clauses are still allowed. The *primal graph* of a CNF formula is a graph that has a node for every variable, and there is an edge between two variables if they coappear in some clause. The *treewidth* of a graph G measures how similar G is to a tree and is defined as the smallest *width* of any *tree decomposition* of G (Robertson and Seymour 1984). The *primal treewidth* of a formula is the treewidth of its primal graph.

Given a CNF formula ϕ, SAT is a decision problem that asks whether there exists a way to assign values to all variables in ϕ such that ϕ evaluates to true. Such a formula is said to be *satisfiable*; otherwise, it is *unsatisfiable*. #SAT is a problem that asks to count the number of such assignments. WMC extends #SAT with a weight function on literals and asks to compute the sum of the weights of the models of the given formula, where the weight of a model is the product of the weights of the literals in it (Chavira and Darwiche 2008). For example, the WMC of the formula $x \lor y$ with a weight function $w\colon \{x, y, \neg x, \neg y\} \to \mathbb{R}$ defined as $w(x) = 0.3$, $w(y) = 0.2$, $w(\neg x) = 0.7$, $w(\neg y) = 0.8$ is $w(x)w(y) + w(x)w(\neg y) + w(\neg x)w(y) = 0.3 \times 0.2 + 0.3 \times 0.8 + 0.7 \times 0.2 = 0.44$.

3 Background on WMC Algorithms

As was mentioned in the introduction, all exact WMC algorithms can be categorised into three groups as those based on search, knowledge compilation, or dynamic programming. CACHET (Sang et al. 2004) is the only search-based algorithm. It is based on the DPLL search procedure (Davis and Putnam 1960; Davis, Logemann, and Loveland 1962) that underlies most SAT algorithms, combined with component caching and clause learning.

Knowledge compilation refers to transformations of propositional formulas into more restrictive formats that make various operations (such as model counting) tractable in the size of the representation (Darwiche and Marquis 2002). C2D (Darwiche 2004), D4 (Lagniez and Marquis 2017), and MINIC2D (Oztok and Darwiche 2015) are all algorithms of this type. C2D compiles to deterministic decomposable negation normal form (d-DNNF) (Darwiche 2001b). Similarly, D4 compiles to decision-DNNF (also known as decomposable decision graphs) (Fargier and Marquis 2006). The only difference between d-DNNF and decision-DNNF is that decision-DNNF has if-then-else constructions instead of disjunctions (Lagniez and Marquis 2017). Finally, MINIC2D (Oztok and Darwiche 2015) compiles to decision-SDDs—a subset of sentential decision diagrams (SDDs) that form a subset of d-DNNF (Darwiche 2011).

All the algorithms mentioned above are #SAT algorithms that were later adapted to support weights. Two recently proposed WMC algorithms instead use data structures that natively support weights and can thus take advantage of redundancies in the numerical values of weights or other numbers. These data structures are representations of *pseudo-Boolean functions*, i.e., functions of the form $f\colon 2^X \to \mathbb{R}$, where X is a set, and 2^X denotes its powerset. ADDMC is the first such algorithm (Dudek, Phan, and Vardi 2020a). It uses ADDs to represent pseudo-Boolean functions, combining and simplifying them in a bottom-up dynamic programming fashion. Since the size of an ADD for f depends on the cardinality of the range of f (Bahar et al. 1997), the performance of the algorithm is sensitive to the numerical values of weights, e.g., to how frequently they repeat. DPMC extends ADDMC in two ways (Dudek, Phan, and Vardi 2020b). First, DPMC allows for the order and nesting of operations on ADDs to be determined from an approximately-minimal-width tree decomposition rather than by heuristics. Second, tensors are offered as an alternative to ADDs.

4 Random k-CNF Formulas with Varying Primal Treewidth

Notation. For any graph G, we write $\mathcal{V}(G)$ for its set of nodes and $\mathcal{E}(G)$ for its set of edges. Let S be a finite set. We write $\mathcal{U}S$ for the discrete uniform probability distribution on S. We represent any other probability distribution as a pair (S, p) where $p\colon S \to [0,1]$ is a probability mass function. For any probability distribution $\mathcal{P}$, we write $x \leftsquigarrow \mathcal{P}$ to denote the act of sampling x from $\mathcal{P}$.

Our random model is based on the following parameters: • the number of variables $\nu \in \mathbb{N}^+$, • density $\mu \in \mathbb{R}_{>0}$, • clause width $\kappa \in \mathbb{N}^+$ (for k-CNF formulas, $\kappa = k$), • a parameter $\rho \in [0,1]$ that influences the primal treewidth of the formula, • the proportion $\delta \in [0,1]$ of variables x such that $w(x) = 1$ and $w(\neg x) = 0$ or $w(x) = 0$ and $w(\neg x) = 1$, • and the proportion $\epsilon \in [0, 1-\delta]$ of variables x such that $w(x) = w(\neg x) = 0.5$. The first three parameters are the standard parameters used to generate random k-CNF formulas with $\nu\mu$ clauses (up to rounding). We expect to observe (possibly different) values of μ that maximize the running time of each algorithm for fixed values of ν and κ. Parameters δ and ϵ control the numerical values of weights and are part of the model because the running time of DPMC (Dudek, Phan, and Vardi 2020b)—and other algorithms based on ADDs—depends on these values. Weights such as zero and one are particularly 'simplifying' because they are respectively the additive and multiplicative identities. Having them

Algorithm 1: Generating a random formula

```
Data: $\nu, \kappa \in \mathbb{N}^+$ such that $\kappa < \nu$, $\mu \in \mathbb{R}_{>0}$, $\rho \in [0,1]$.
Result: A $k$-CNF formula $\phi$.
1  $\phi \leftarrow$ empty CNF formula;
2  $G \leftarrow$ empty graph;
3  for $i \leftarrow 1$ to $\lfloor \nu\mu \rfloor$ do
4      $X \leftarrow \varnothing$;
5      for $j \leftarrow 1$ to $\kappa$ do
6          $x \leftarrow$ NewVariable($X$, $G$);
7          $\mathcal{V}(G) \leftarrow \mathcal{V}(G) \cup \{x\}$;
8          $\mathcal{E}(G) \leftarrow \mathcal{E}(G) \cup \{\{x, y\} \mid y \in X\}$;
9          $X \leftarrow X \cup \{x\}$;
10     $\phi \leftarrow \phi \cup \{l \leftsquigarrow \mathcal{U}\{x, \neg x\} \mid x \in X\}$;
11 return $\phi$;
12 Function NewVariable($X$, $G$):
13     $N \leftarrow \{e \in \mathcal{E}(G) \mid |e \cap X| = 1\}$;
14     if $N = \varnothing$ then
15         return $x \leftsquigarrow \mathcal{U}(\{x_1, x_2, \dots, x_\nu\} \setminus X)$;
16     return $x \leftsquigarrow \Big(\{x_1, x_2, \dots, x_\nu\} \setminus X,$
17         $y \mapsto \frac{1-\rho}{\nu-|X|} + \rho\frac{|\{z \in X \mid \{y,z\} \in \mathcal{E}(G)\}|}{|N|}\Big)$;
```

propagate through the algorithm reduces the size of many ADDs used by DPMC, making the algorithm more efficient. Including many copies of the same weight (e.g., 0.5) can similarly simplify ADDs as well. Other WMC algorithms are indifferent to the numerical values of weights.

The process behind generating random k-CNF formulas is summarized as Algorithm 1. For the rest of this section, let $x_1, x_2, \dots, x_\nu$ be the variables of the formula under construction. We simultaneously construct both formula ϕ and its primal graph G.[6] Each iteration of the first for-loop adds a clause to ϕ. This is done by constructing a set X of variables to be included in the clause, and then randomly adding either x or $\neg x$ to the clause for each $x \in X$ on Line 10. Function NewVariable randomly selects each new variable x, and Lines 7 to 9 add x to the graph and the formula while also adding edges between x and all the other variables in the clause. To select each variable, Line 13 defines set N to contain all edges with exactly one endpoint in X. The edges that will be added to G by Line 8 will form a subset of N. If $N = \varnothing$, we select the variable uniformly at random (u.a.r.) from all viable candidates. Otherwise, ρ determines how much we bias the uniform distribution towards variables that would introduce the smallest number of new edges to G.

When $\rho = 0$, Algorithm 1 reduces to what has become the standard random model for k-CNF formulas. Equivalently to Franco and Paull (1983), we independently sample a fixed number of clauses, each clause has no duplicate variables, and each variable becomes either a positive or a negative

[6]The idea to directly take the primal graph into consideration while generating the formula is new—cf. random SAT instance generators based on, e.g., adversarial evolution (Hossain et al. 2010) and community structure (Giráldez-Cru and Levy 2016).

literal with equal probabilities. At the other extreme, when $\rho = 1$, the first variable of a clause is still chosen u.a.r., but all other variables are chosen from those that already coappear in a clause (if possible). The probability that a variable is selected to be included in a clause scales linearly w.r.t. the proportion of edges in N that would be repeatedly added to G if the variable y was added to the clause. This is an arbitrary choice (which appears to work well, see Section 4.1) although alternatives (e.g., exponential scaling) could be considered. As long as $\rho < 1$, every k-CNF formula retains a positive probability of being generated by the algorithm.

To transform the generated formula into a WMC instance, we need to define weights on literals.[7] We want to partition all variables into three groups: those with weights equal to zero and one, those with weights equal to 0.5, and those with arbitrary weights, where the size of each group is determined by δ and ϵ. To do this, we sample a permutation $\pi \leftsquigarrow \mathcal{U}S_\nu$ (where S_ν is the permutation group on $\{1, 2, \dots, \nu\}$), and assign to each *variable* x_n a weight drawn u.a.r. from • $\mathcal{U}\{\,0,1\,\}$ if $\pi(n) \le \nu\delta$, • $\mathcal{U}\{\,0.5\,\}$ if $\nu\delta < \pi(n) \le \nu\delta + \nu\epsilon$, • and $\mathcal{U}\{\,0.01, 0.02, \dots, 0.99\,\}$[8] if $\pi(n) > \nu\delta + \nu\epsilon$. We extend these weights to weights on *literals* by choosing the weight of each positive literal to be equal to the weight of its variable, and the weight of each negative literal to be such that $w(x) + w(\neg x) = 1$ for all variables x. This restriction is to ensure consistent answers among the algorithms.

Example 1. Let $\nu = 5$, $\mu = 0.6$, $\kappa = 3$, $\rho = 0.3$, $\delta = 0.4$, and $\epsilon = 0.2$ and consider how Algorithm 1 generates a random instance. Since $\kappa = 3$, and $\lfloor \nu\mu \rfloor = 3$, the algorithm will generate a 3-CNF formula with three clauses.

For the first variable of the first clause, we are choosing u.a.r. from $\{\,x_1, x_2, \dots, x_5\,\}$. Suppose the algorithm chooses x_5. Graph G then gets its first node but no edges. The second variable is chosen u.a.r. from $\{\,x_1, x_2, x_3, x_4\,\}$. Suppose the second variable is x_2. Then G gets another node and its first edge between x_2 and x_5. The third variable in the first clause is similarly chosen u.a.r. from $\{\,x_1, x_3, x_4\,\}$ because the only edge in G has both endpoints in $X = \{\,x_2, x_5\,\}$, and so $N = \emptyset$. Suppose the third variable is x_1. Graph G becomes a triangle connecting x_1, x_2, and x_5. Each of the three variables is then added to the clause as either a positive or a negative literal (with equal probabilities). Thus, the first clause becomes, e.g., $\neg x_5 \lor x_2 \lor x_1$.

The first variable of the second clause is chosen u.a.r. from $\{\,x_1, x_2, \dots, x_5\,\}$. Suppose it is x_5 again. When the function `NewVariable` tries to choose the second variable, $X = \{\,x_5\,\}$, and so $N = \{\,\{\,x_1, x_5\,\}, \{\,x_2, x_5\,\}\,\}$. The second variable is chosen from the discrete probability distribution

$$\Pr(x_1) = \Pr(x_2) = \frac{1 - 0.3}{5 - 1} + 0.3 \times \frac{1}{2} = 0.325$$

and

$$\Pr(x_3) = \Pr(x_4) = \frac{1 - 0.3}{5 - 1} = 0.175.$$

We skip the details of how all remaining variables and clauses are selected and consider the weight assignment. First, we shuffle the list of variables and get, e.g., $L = (x_4, x_3, x_2, x_1, x_5)$. This means that the first $\nu\delta = 5 \times 0.4 = 2$ variables of L get weights u.a.r. from $\{\,0, 1\,\}$, the next $\nu\epsilon = 5 \times 0.2 = 1$ variable gets a weight of 0.5, and the remaining two variables get weights u.a.r. from $\{\,0.01, 0.02, \dots, 0.99\,\}$. The weight function $w\colon \{\,x_1, x_2, \dots, x_5, \neg x_1, \neg x_2, \dots, \neg x_5\,\} \to [0, 1]$ can then be defined as, e.g., $w(x_4) = w(\neg x_3) = 0$, $w(x_3) = w(\neg x_4) = 1$, $w(x_2) = w(\neg x_2) = 0.5$, $w(x_1) = 0.23$, $w(\neg x_1) = 0.77$, $w(x_5) = 0.18$, and $w(\neg x_5) = 0.82$.

4.1 Validating the Model

The idea behind our model is that manipulating the value of ρ should allow us to generate instances of varying primal treewidth. Is this effect observable in practice? In addition, as WMC instances are mostly used for probabilistic inference, they tend to be satisfiable. Therefore, we want to filter out unsatisfiable instances from those generated by the model and need to ensure that the proportion of satisfiable instances remains sufficiently high. Given that higher values of ρ can result in constraints on variables being more localised and concentrated, we ask: are instances generated with higher values of ρ less likely to be satisfiable? To answer both questions, we run the following experiment.

Experiment 1. Fix $\nu = 100$, $\delta = \epsilon = 0$, and consider random instances with $\mu = 2.5 \times \sqrt{2}^{-5}, 2.5 \times \sqrt{2}^{-4}, \dots, 2.5 \times \sqrt{2}^{5}$, $\kappa = 2, 3, 4, 5$, and ρ going from 0 to 1 in steps of 0.01. For each combination of parameters, we generate ten instances. We check if each instance is satisfiable using MiniSat[9] 2.2.0 (Eén and Sörensson 2003) and calculate its (approximate) primal treewidth using HTD (Abseher, Musliu, and Woltran 2017).

Figure 1 shows the relationship between ρ and primal treewidth. Except for when both μ and κ are set to very low values (i.e., the formulas are small in both clause width and the number of clauses), primal treewidth decreases as ρ increases. This downward trend becomes sharper as μ increases, however, not uniformly: it splits into a roughly linear segment that approaches a horizontal line (for most values of ρ) and a sharply-decreasing segment that approaches a vertical line (when ρ is close to one). Higher values of κ seem to expedite this transition, i.e., with a higher value of κ, a lower value of μ is sufficient for a smooth downward curve between ρ and primal treewidth to turn into a combination of a horizontal and a vertical line. While this behaviour may be troublesome when generating formulas with higher values of μ (almost all of which would be unsatisfiable), the relationship between ρ and primal treewidth is excellent for generating 3-CNF formulas close to and below the satisfiability threshold of 4.25 (Crawford and Auton 1996).

Regarding satisfiability, the proportion of satisfiable 3-CNF formulas drops from 63.6 % when $\rho = 0$ to 50.9 % when $\rho = 1$, so—while ρ does affect satisfiability—the effect is not significant enough to influence our experimental

[7]Note that algorithms such as DPMC and ADDMC (Dudek, Phan, and Vardi 2020a,b) support a more flexible way of assigning weights that can lead to significant performance improvements (Dilkas and Belle 2021a,b).

[8]For convenience, we represent $(0, 1)$ as 99 discrete values.

[9]http://minisat.se/MiniSat.html

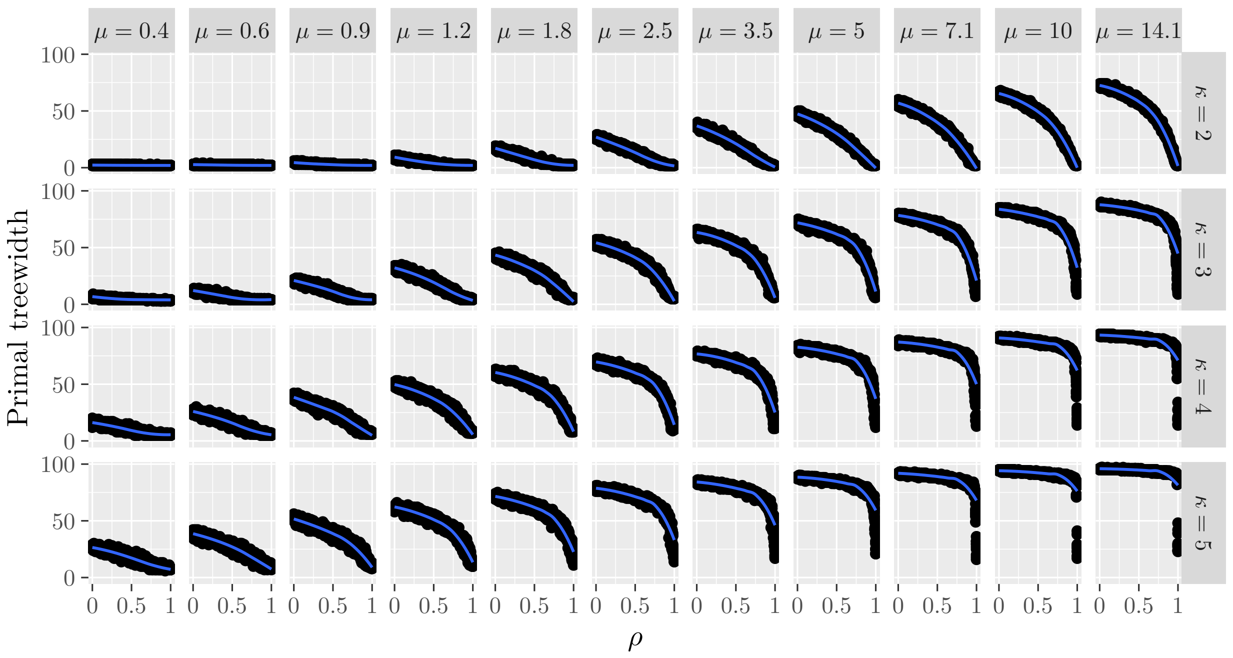


Figure 1: The relationship between ρ and primal treewidth for various values of μ and κ for k-CNF formulas from Experiment 1. Black points represent individual instances, and blue lines are smoothed means computed using locally weighted smoothing. The values of μ are rounded to one decimal place.

setup in the next section.

5 Experimental Results

In this section, we describe three experiments that examine how the running times of WMC algorithms change w.r.t. parameters of our random model. All experiments were run on Intel Xeon E5–2630 with Scientific Linux 7, GCC 10.2.0, Python 3.8.1, R 4.1.0, C2D 2.20 (Darwiche 2004), CACHET 1.22 (Sang et al. 2004), and HTD 1.2.0 (Abseher, Musliu, and Woltran 2017). With both C2D and D4, we use QUERY-DNNF[10] to compute the numerical answer from the compiled circuit. We omit ADDMC (Dudek, Phan, and Vardi 2020a) from our experiments as it exceeds time and memory limits on too many instances; however, observations about the behaviour of DPMC (Dudek, Phan, and Vardi 2020b) apply to ADDMC as well, with the addendum that the tree decomposition implicitly used by ADDMC may have a significantly higher width. DPMC is run with tree decomposition-based planning and ADD-based execution—the combination that was originally found to be most effective (Dudek, Phan, and Vardi 2020b). The tree decomposition is found using one iteration of HTD[11] (Abseher, Musliu, and Woltran 2017). We restrict our attention to 3-CNF formulas, generate 100 satisfiable instances for each combination of parameters, and run each of the five algorithms with a 500 s time limit and an 8 GiB memory limit. While both limits are somewhat low, we prioritise large numbers of instances to increase the accuracy and reliability of our results. Unless stated otherwise, in each plot of this section, lines denote median values, and shaded regions show interquartile ranges. We run the following three experiments, setting $\nu = 70$ in all of them as we found that this produces instances of suitable difficulty.

Experiment 2 (Density and Primal Treewidth)**.** Let $\nu = 70$, μ go from 1 to 4.3 in steps of 0.3, ρ go from 0 to 0.5 in steps of 0.01, and $\delta = \epsilon = 0$.

Experiment 3 (δ)**.** Let $\nu = 70$, $\mu = 2.2$[12], $\rho = 0$, δ go from 0 to 1 in steps of 0.01, and $\epsilon = 0$.

Experiment 4 (ϵ)**.** Same as Experiment 3 but with $\delta = 0$ and ϵ going from 0 to 1 in steps of 0.01.

In each experiment, the proportion of algorithm runs that timed out never exceeded 3.8 %. While in Experiment 2 only 1 % of experimental runs ran out of memory, the same percentage was higher in Experiments 3 and 4—10 and 12 %, respectively. D4 (Lagniez and Marquis 2017) and C2D (Darwiche 2004) are the algorithms that experienced the most issues fitting within the memory limit, accounting for 66–72 % and 28–33 % of such instances, respectively. We exclude the runs that terminated early due to running out of memory from the rest of our analysis.

In Experiment 2, we investigate how the running time of each algorithm depends on the density and primal treewidth by varying both μ and ρ. The results are in Fig. 2. The first

[10]http://www.cril.univ-artois.fr/kc/d-DNNF-reasoner.html

[11]https://github.com/mabseher/htd

[12]Experiment 2 shows this density to be the most challenging for DPMC (Dudek, Phan, and Vardi 2020b).

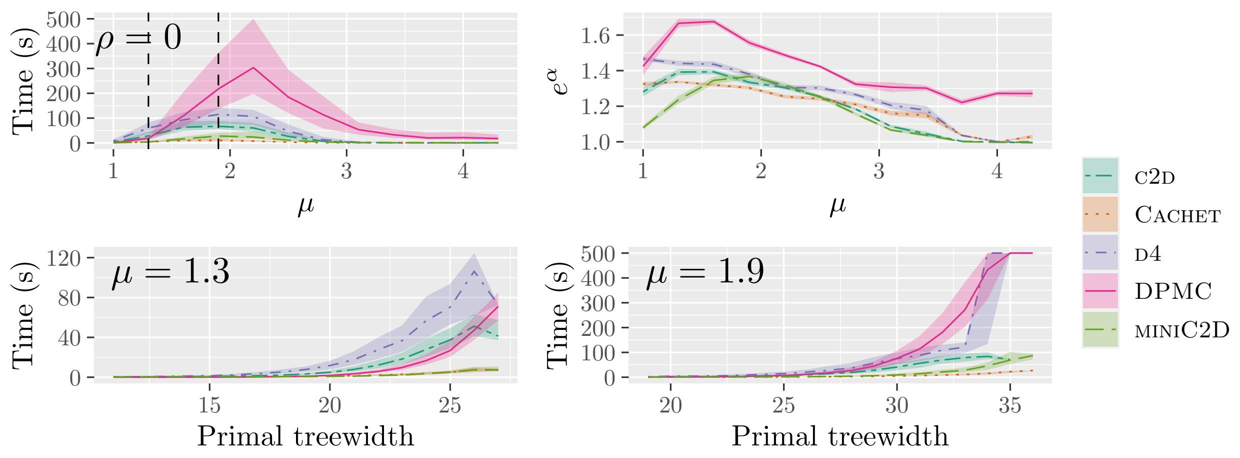
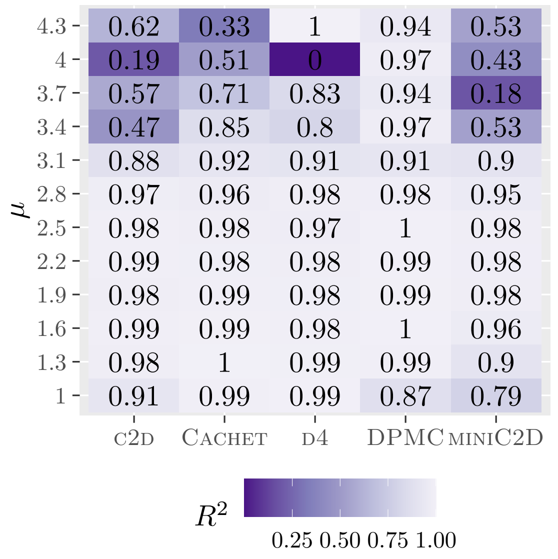


Figure 2: Visualisations of the data from Experiment 2. The top-left plot shows how the running time of each algorithm changes w.r.t. density when $\rho = 0$. For each algorithm and value of μ, each line in the top-right plot shows the estimated base of the exponential for a linear model where running time is assumed to scale exponentially w.r.t. primal treewidth, and shaded regions show standard error. The two plots at the bottom show changes in the running time of each algorithm w.r.t. primal treewidth for selected fixed values of μ. These values are also marked by dashed vertical lines in the top-left plot.



μ	C2D	CACHET	D4	DPMC	MINIC2D
4.3	0.62	0.33	1	0.94	0.53
4	0.19	0.51	0	0.97	0.43
3.7	0.57	0.71	0.83	0.94	0.18
3.4	0.47	0.85	0.8	0.97	0.53
3.1	0.88	0.92	0.91	0.91	0.9
2.8	0.97	0.96	0.98	0.98	0.95
2.5	0.98	0.98	0.97	1	0.98
2.2	0.99	0.98	0.98	0.99	0.98
1.9	0.98	0.99	0.98	0.99	0.98
1.6	0.99	0.99	0.98	1	0.96
1.3	0.98	1	0.99	0.99	0.9
1	0.91	0.99	0.99	0.87	0.79

Figure 3: The coefficients of determination (rounded to one decimal place) of all the linear models fitted for the top-right subplot of Fig. 2

thing to note is that the peak hardness w.r.t. density occurs at around 1.9 for all algorithms except for DPMC (Dudek, Phan, and Vardi 2020b), which peaks at 2.2 instead. This finding is consistent with previous work, which shows CACHET to peak at 1.8 (Sang et al. 2004).[13]

The other question we want to investigate using this experiment is how each algorithm scales w.r.t. primal treewidth. The two plots at the bottom of Fig. 2 show this relationship for fixed values of μ, and one can see some evidence that the running time of DPMC (Dudek, Phan, and Vardi 2020b) grows faster w.r.t. primal treewidth than the running time of the other algorithms. The top-right subplot of Fig. 2 shows how this growth depends on μ. For each algorithm and value of μ in Experiment 2, we select the median runtime for all available values of primal treewidth and fit the model $\ln t \sim \alpha w + \beta$, where t is the median running time of the algorithm, w is the primal treewidth, and α and β are parameters.[14] In other words, this model attempts to express median running time as $e^{\beta}(e^{\alpha})^{w}$. We find that, indeed, DPMC scales worse w.r.t. primal treewidth than any other algorithm across all values of μ and is the only algorithm that does not become indifferent to primal treewidth when faced with high-density formulas. A second look at the top-left subplot of Fig. 2 suggests an explanation for the latter observation. The running times of all algorithms except for DPMC approach zero when $\mu > 3$ while the median running time of DPMC approaches a small non-zero constant instead. This observation also explains why Fig. 3 shows that the fitted models fail to explain the data for non-ADD algorithms running on high-density instances—the running times are too small to be meaningful. In all other cases, an exponential relationship between primal treewidth and runtime fits the experimental data remarkably well.

Another thing to note is that MINIC2D (Oztok and Darwiche 2015) is the only algorithm that exhibits a clear low-high-low pattern in the top right subplot of Fig. 2. To a smaller extent, the same may apply to C2D (Darwiche 2004) and DPMC (Dudek, Phan, and Vardi 2020b) as well, al-

[13]For comparison, #SAT algorithms have been observed to peak at densities 1.2 and 1.5 (Bayardo Jr. and Pehoushek 2000).

[14]Similar statistical analyses have been used to investigate polynomial-to-exponential phase transitions in SAT (Coarfa et al. 2003) and the behaviour of SAT solvers on CNF-XOR formulas (Dudek, Meel, and Vardi 2017).

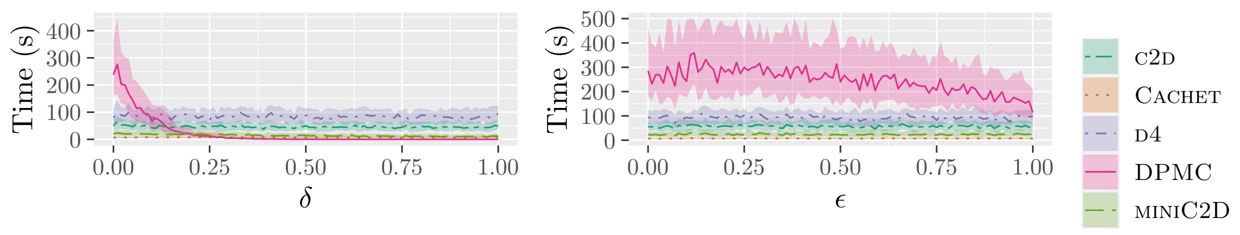


Figure 4: Changes in the running time of each algorithm as a result of changing δ (on the left-hand side) and ϵ (on the right-hand side) according to the data from Experiments 3 and 4

though the evidence for this is limited due to relatively large gaps between different values of μ in Experiment 2. In contrast, the running times of CACHET (Sang et al. 2004) and D4 (Lagniez and Marquis 2017) remain dependent on primal treewidth even when the density of the WMC instance is very low, suggesting that MINIC2D should have an advantage on low-density high-primal-treewidth instances.

Experiments 3 and 4 investigate how changing the numerical values of weights can simplify a WMC instance. The results are in Fig. 4. As expected, the running time of all algorithms other than DPMC (Dudek, Phan, and Vardi 2020b) stay the same regardless of the value of δ or ϵ. The running time of DPMC, however, experiences a sharp (exponential?) decline with increasing δ. The decline w.r.t. ϵ is also present, although significantly less pronounced and with high variance.

How are these random instances different from real data? As a representative sample, we take the WMC encodings of Bayesian networks created using the method by Sang, Beame, and Kautz (2005b) as found in the experimental setup[15] of the DPMC paper (Dudek, Phan, and Vardi 2020b). A typical real WMC instance has $\nu = 200$ variables, half of which have equal weights (i.e., $\epsilon = 0.5$), an average clause width of $\kappa = 2.6$, a density of $\mu = 2.5$, and a primal treewidth of 28. Our random instances have fewer variables and (for the most part) lower density. Another important difference is that our instances are in k-CNF whereas a typical encoding of a Bayesian network has many two-literal clauses mixed with clauses of various longer widths. Despite real instances having more variables, their primal treewidth is rather low. Perhaps this partially explains why the performance of DPMC is in line with the performance of all other algorithms on traditionally-used benchmarks (Dudek, Phan, and Vardi 2020b) despite struggling with most of our random data.

To sum, we found that C2D (Darwiche 2004) and D4 (Lagniez and Marquis 2017) are the most memory-intensive algorithms, CACHET (Sang et al. 2004) is great on random instances in general, MINIC2D (Oztok and Darwiche 2015) exceeds on low-density high-primal-treewidth instances, and DPMC (Dudek, Phan, and Vardi 2020b) is at its best on low-density low-primal-treewidth instances. Furthermore, a median instance with all weights equal to each other is about three times easier for DPMC than a median instance with random weights. Another important observation is about how peak hardness w.r.t. density depends on the algorithm: DPMC peaks at a higher density than all other WMC algorithms, which peak at a higher density than (some) #SAT algorithms.

[15]https://github.com/vardigroup/DPMC/releases

6 Conclusions and Future Work

In this paper, we studied the behaviour of and differences among WMC algorithms on random instances generated by a standard model for k-CNF formulas extended with parameters that control primal treewidth and literal weights. Among other things, we established statistical evidence for the existence of an exponential relationship between primal treewidth and the running time of all WMC algorithms. The running time of ADD-based algorithms was observed to peak at a higher density, scale worse w.r.t. primal treewidth, and depend negatively on repeating weight values compared to algorithms based on search or knowledge compilation. These observations can, to some degree, be extended to a closely related weighted projected model counting algorithm (Dudek, Phan, and Vardi 2021) as well as to other applications of ADDs more generally, e.g., probabilistic inference (Chavira and Darwiche 2007; Gogate and Domingos 2011) and stochastic planning (Hoey et al. 1999).

One limitation of our work is that variability in primal treewidth was achieved via a parameter, and this could bias randomness in some unexpected way (although it is encouraging that there is only a slight decrease in the proportion of satisfiable instances between $\rho = 0$ and $\rho = 1$). Perhaps a theoretical investigation of the proposed model is warranted, including a characterisation of how ρ influences primal treewidth and the structure of the primal graph more generally. Since treewidth is widely used in parameterised complexity (Downey and Fellows 2013), formally establishing a connection with ρ could make our random model useful for a variety of other hard computational problems.

To keep the number of experiments feasible, we restricted our attention to 3-CNF formulas, although, of course, this is not very representative of real-world WMC instances. The model could be adapted to generate non-k-CNF formulas, and perhaps a more representative structure could be achieved by introducing new variables that clauses define to be equivalent to select conjunctions of literals as is done in one of the WMC encodings for Bayesian networks (Darwiche 2002).

References

Abseher, M.; Musliu, N.; and Woltran, S. 2017. htd - A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *CPAIOR*, volume 10335 of *Lecture Notes in Computer Science*, 376–386. Springer.

Achlioptas, D.; and Moore, C. 2002. The Asymptotic Order of the Random k -SAT Threshold. In *FOCS*, 779–788. IEEE Computer Society.

Ansótegui, C.; Bonet, M. L.; and Levy, J. 2009. Towards Industrial-Like Random SAT Instances. In *IJCAI*, 387–392.

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2009. Solving #SAT and Bayesian Inference with Backtracking Search. *J. Artif. Intell. Res.*, 34: 391–442.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods Syst. Des.*, 10(2/3): 171–206.

Bart, A.; Koriche, F.; Lagniez, J.; and Marquis, P. 2016. An Improved CNF Encoding Scheme for Probabilistic Inference. In *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, 613–621. IOS Press.

Bayardo Jr., R. J.; and Pehoushek, J. D. 2000. Counting Models Using Connected Components. In *AAAI/IAAI*, 157–162. AAAI Press / The MIT Press.

Belle, V. 2017. Open-Universe Weighted Model Counting. In *AAAI*, 3701–3708. AAAI Press.

Belle, V.; Passerini, A.; and Van den Broeck, G. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *IJCAI*, 2770–2776. AAAI Press.

Bläsius, T.; Friedrich, T.; and Sutton, A. M. 2019. On the Empirical Time Complexity of Scale-Free 3-SAT at the Phase Transition. In *TACAS (1)*, volume 11427 of *Lecture Notes in Computer Science*, 117–134. Springer.

Chakraborty, S.; Fried, D.; Meel, K. S.; and Vardi, M. Y. 2015. From Weighted to Unweighted Model Counting. In *IJCAI*, 689–695. AAAI Press.

Chavira, M.; and Darwiche, A. 2005. Compiling Bayesian Networks with Local Structure. In *IJCAI*, 1306–1312. Professional Book Center.

Chavira, M.; and Darwiche, A. 2006. Encoding CNFs to Empower Component Analysis. In *SAT*, volume 4121 of *Lecture Notes in Computer Science*, 61–74. Springer.

Chavira, M.; and Darwiche, A. 2007. Compiling Bayesian Networks Using Variable Elimination. In *IJCAI*, 2443–2449.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7): 772–799.

Chavira, M.; Darwiche, A.; and Jaeger, M. 2006. Compiling relational Bayesian networks for exact inference. *Int. J. Approx. Reason.*, 42(1-2): 4–20.

Coarfa, C.; Demopoulos, D. D.; Aguirre, A. S. M.; Subramanian, D.; and Vardi, M. Y. 2003. Random 3-SAT: The Plot Thickens. *Constraints An Int. J.*, 8(3): 243–261.

Coja-Oghlan, A.; and Wormald, N. 2018. The Number of Satisfying Assignments of Random Regular k-SAT Formulas. *Comb. Probab. Comput.*, 27(4): 496–530.

Crawford, J. M.; and Auton, L. D. 1996. Experimental Results on the Crossover Point in Random 3-SAT. *Artif. Intell.*, 81(1-2): 31–57.

Dal, G. H.; Laarman, A. W.; and Lucas, P. J. F. 2018. Parallel Probabilistic Inference by Weighted Model Counting. In *PGM*, volume 72 of *Proceedings of Machine Learning Research*, 97–108. PMLR.

Darwiche, A. 2001a. Decomposable negation normal form. *J. ACM*, 48(4): 608–647.

Darwiche, A. 2001b. On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *J. Appl. Non Class. Logics*, 11(1-2): 11–34.

Darwiche, A. 2002. A Logical Approach to Factoring Belief Networks. In *KR*, 409–420. Morgan Kaufmann.

Darwiche, A. 2004. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *ECAI*, 328–332. IOS Press.

Darwiche, A. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *IJCAI*, 819–826. IJCAI/AAAI.

Darwiche, A.; and Marquis, P. 2002. A Knowledge Compilation Map. *J. Artif. Intell. Res.*, 17: 229–264.

Davis, M.; Logemann, G.; and Loveland, D. W. 1962. A machine program for theorem-proving. *Commun. ACM*, 5(7): 394–397.

Davis, M.; and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3): 201–215.

Dilkas, P.; and Belle, V. 2020. Generating Random Logic Programs Using Constraint Programming. In *CP*, volume 12333 of *Lecture Notes in Computer Science*, 828–845. Springer.

Dilkas, P.; and Belle, V. 2021a. Weighted Model Counting with Conditional Weights for Bayesian Networks. In *UAI*, Proceedings of Machine Learning Research. AUAI Press.

Dilkas, P.; and Belle, V. 2021b. Weighted Model Counting Without Parameter Variables. In *SAT*, volume 12831 of *Lecture Notes in Computer Science*, 134–151. Springer.

Downey, R. G.; and Fellows, M. R. 2013. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer.

Dudek, J. M.; Meel, K. S.; and Vardi, M. Y. 2017. The Hard Problems Are Almost Everywhere For Random CNF-XOR Formulas. In *IJCAI*, 600–606. ijcai.org.

Dudek, J. M.; Phan, V.; and Vardi, M. Y. 2020a. ADDMC: Weighted Model Counting with Algebraic Decision Diagrams. In *AAAI*, 1468–1476. AAAI Press.

Dudek, J. M.; Phan, V. H. N.; and Vardi, M. Y. 2020b. DPMC: Weighted Model Counting by Dynamic Programming on Project-Join Trees. In *CP*, volume 12333 of *Lecture Notes in Computer Science*, 211–230. Springer.

Dudek, J. M.; Phan, V. H. N.; and Vardi, M. Y. 2021. ProCount: Weighted Projected Model Counting with Graded Project-Join Trees. In *SAT*, volume 12831 of *Lecture Notes in Computer Science*, 152–170. Springer.

Eén, N.; and Sörensson, N. 2003. An Extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Fargier, H.; and Marquis, P. 2006. On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae. In *AAAI*, 42–47. AAAI Press.

Fichte, J. K.; Hecher, M.; Woltran, S.; and Zisser, M. 2018. Weighted Model Counting on the GPU by Exploiting Small Treewidth. In *ESA*, volume 112 of *LIPIcs*, 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D. S.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory Pract. Log. Program.*, 15(3): 358–401.

Franco, J.; and Paull, M. C. 1983. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discret. Appl. Math.*, 5(1): 77–87.

Galanis, A.; Goldberg, L. A.; Guo, H.; and Yang, K. 2020. Counting Solutions to Random CNF Formulas. In *ICALP*, volume 168 of *LIPIcs*, 53:1–53:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Giráldez-Cru, J.; and Levy, J. 2016. Generating SAT instances with community structure. *Artif. Intell.*, 238: 119–134.

Giráldez-Cru, J.; and Levy, J. 2017. Locality in Random SAT Instances. In *IJCAI*, 638–644. ijcai.org.

Gogate, V.; and Domingos, P. M. 2011. Approximation by Quantization. In *UAI*, 247–255. AUAI Press.

Gogate, V.; and Domingos, P. M. 2016. Probabilistic theorem proving. *Commun. ACM*, 59(7): 107–115.

Hoey, J.; St-Aubin, R.; Hu, A. J.; and Boutilier, C. 1999. SPUDD: Stochastic Planning using Decision Diagrams. In *UAI*, 279–288. Morgan Kaufmann.

Holtzen, S.; Van den Broeck, G.; and Millstein, T. D. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA): 140:1–140:31.

Hossain, M. M.; Abbass, H. A.; Lokan, C.; and Alam, S. 2010. Adversarial Evolution: Phase transition in non-uniform hard satisfiability problems. In *IEEE Congress on Evolutionary Computation*, 1–8. IEEE.

Lagniez, J.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *IJCAI*, 667–673. ijcai.org.

Mitchell, D. G.; Selman, B.; and Levesque, H. J. 1992. Hard and Easy Distributions of SAT Problems. In *AAAI*, 459–465. AAAI Press / The MIT Press.

Oztok, U.; and Darwiche, A. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *IJCAI*, 3141–3148. AAAI Press.

Pote, Y.; Joshi, S.; and Meel, K. S. 2019. Phase Transition Behavior of Cardinality and XOR Constraints. In *IJCAI*, 1162–1168. ijcai.org.

Purdom Jr., P. W.; and Brown, C. A. 1983. An Analysis of Backtracking with Search Rearrangement. *SIAM J. Comput.*, 12(4): 717–733.

Renkens, J.; Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2014. Explanation-Based Approximate Weighted Model Counting for Probabilistic Logics. In *AAAI*, 2490–2496. AAAI Press.

Riguzzi, F. 2020. Quantum Weighted Model Counting. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 2640–2647. IOS Press.

Robertson, N.; and Seymour, P. D. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1): 49–64.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*.

Sang, T.; Beame, P.; and Kautz, H. A. 2005a. Heuristics for Fast Exact Model Counting. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, 226–240. Springer.

Sang, T.; Beame, P.; and Kautz, H. A. 2005b. Performing Bayesian Inference by Weighted Model Counting. In *AAAI*, 475–482. AAAI Press / The MIT Press.

Van den Broeck, G.; Taghipour, N.; Meert, W.; Davis, J.; and De Raedt, L. 2011. Lifted Probabilistic Inference by First-Order Knowledge Compilation. In *IJCAI*, 2178–2185. IJCAI/AAAI.

Xu, J.; Zhang, Z.; Friedman, T.; Liang, Y.; and Van den Broeck, G. 2018. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, 5498–5507. PMLR.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.*, 32: 565–606.