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

Paulius Dilkas

University of Edinburgh, Edinburgh, UK
p.dilkas@sms.ed.ac.uk

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) [16]—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 [7, 13, 14, 25, 65], probabilistic programs [49], and probabilistic logic programs [40]. More recently, WMC was used in the context of neural-symbolic artificial intelligence as well [68]. Extensions of WMC add support for continuous variables [10], infinite domains [9], and first-order logic [67, 46] and generalise the definition to support arbitrary pseudo-Boolean functions instead of clauses [31]. Exact WMC algorithms can be broadly classified as based on search [63, 66], knowledge compilation [26, 52, 55], and dynamic programming [34, 35]. Other alternatives include approximate [59] and parallel algorithms [21,

39], hybrid approaches [47], quantum computing [60], and reduction to model counting [12].

Recent papers that include experimental comparisons of WMC algorithms show many of them performing very similarly overall [34, 35] but with overwhelming differences when run on specific subsets of data [30, 31, 52]. Examples of such segregating data sets include bipartite Bayesian networks by Sang et al. [65] and relational Bayesian networks by Chavira et al. [17] 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 [69]. 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 [50].

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 [41, 57] continue to be instrumental up to this day for, e.g., establishing the location of the threshold between satisfiable and unsatisfiable instances [2] and efficiently approximating #SAT [42]. Other random models consider non-uniform variable frequencies [3], fixing the number of times each variable occurs both positively and negatively [19], and adding other constraints such as cardinality and 'exclusive or' [56]. In contrast, only one WMC algorithm so far has been analysed using random instances [63, 64]. Similarly, while there is a recent attempt [29] 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 [53]. Later work revealed that the interaction between density and empirical hardness is much more solver-dependent [18]. Many other parameters such as heterogeneity, locality, and modularity have emerged from attempts to generate random instances similar to industry benchmarks for SAT [3, 11, 43, 44].

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) [4, 23, 26, 63]. 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) [5] is also known to depend on the numerical values of weights [34, 35]. 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] [26], CACHET[2] [63], D4[3] [52], DPMC[4] [35], and MINIC2D[5] [55]—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.

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 [61]. 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 [16]. For example, the WMC of the

[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/

formula $x \vee y$ with a weight function $w\colon \{ x, y, \neg x, \neg y \} \to \mathbb{R}_{\geq 0}$ 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

In this section, we briefly review the three major approaches to WMC—search, knowledge compilation, and dynamic programming—and their corresponding algorithms. The main search-based WMC algorithm Cachet [63] is based on a conflict-driven clause learning SAT solver [54], which is then extended with a component caching scheme and adapted to counting.

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 [28]. c2d [26], d4 [52], and miniC2D [55] are all algorithms of this type. c2d compiles to deterministic decomposable negation normal form (d-DNNF) [24]. Similarly, d4 compiles to decision-DNNF (also known as decomposable decision graphs) [38]. The only difference between d-DNNF and decision-DNNF is that decision-DNNF has if-then-else constructions instead of disjunctions [52]. Finally, miniC2D compiles to decision-SDDs—a subset of sentential decision diagrams (SDDs) that form a subset of d-DNNF [27].

All of the algorithms mentioned above execute in exactly the same way regardless of whether computing WMC or #SAT. Two recent WMC algorithms instead use data structures whose size (and thus the runtime of the algorithm) depends on the numerical values of weights. These data structures are representations of *pseudo-Boolean functions*, i.e., functions of the form $f\colon 2^X \to \mathbb{R}_{\geq 0}$, where X is a set, and 2^X denotes its powerset. ADDMC is the first such algorithm [34]. 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 [5], 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 [35]. 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.[6] Second, tensors are offered as an alternative to ADDs.

In all known parameterised complexities of WMC algorithms, the exponential factor is a function of primal treewidth or a closely related parameter. Interestingly, c2d is specifically designed to handle high primal treewidth (which the author refers to as *connectivity* [22]) and improves upon an earlier algorithm that has $\mathcal{O}(mw2^w)$ time complexity, where m is the number of clauses, and w is the width of the decomposition tree which is known to be at most primal treewidth [23, 26]. While the complexity of Cachet was not analysed directly, the algorithm is based on component caching which is known to have a $2^{\mathcal{O}(w)}n^{\mathcal{O}(1)}$ time complexity, where n is the number of variables, and w is the branchwidth of the underlying

[6] There is also a recent line of work in using tree decompositions to guide the heuristics of search-based model counters [51].

hypergraph [4, 63], which is known to be within a constant factor of primal treewidth [62]. Similarly, the complexity of DPMC is not described in the paper, although the authors define a notion of width w that is at most primal treewidth plus one and estimate the running time of the (execution part of the) algorithm to be proportional to 2^w [35].

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 $\mathcal{P}$. For instance, $x \leftsquigarrow (\{1, 2\}, \{1 \mapsto 0.1, 2 \mapsto 0.9\})$ means that x becomes equal to 1 with probability 0.1 or to 2 with probability 0.9.

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 [35]—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 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 $x_1, x_2, \ldots, x_\nu$ be the variables of the formula under construction. We simultaneously construct both formula ϕ and its primal graph G.[7] 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

[7] 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 [50] and community structure [43].

Algorithm 1: Generating a random formula

Input: $\nu, \kappa \in \mathbb{N}^+$ such that $\kappa < \nu$, $\mu \in \mathbb{R}_{>0}$, $\rho \in [0, 1]$.
Output: A k-CNF formula ϕ.

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** κ **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** ϕ;
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** $x \leftsquigarrow \mathcal{U}(\{ x_1, x_2, \ldots, x_\nu \} \setminus X)$;
16 **return** $x \leftsquigarrow \Big(\{ x_1, x_2, \ldots, x_\nu \} \setminus X,$
17 $y \mapsto \frac{1-\rho}{\nu - |X|} + \rho \frac{|\{ z \in X \mid \{ y, z \} \in \mathcal{E}(G) \}|}{|N|} \Big)$;

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 [41], we independently sample a fixed number of clauses, each clause has no duplicate variables, and each variable becomes either a positive or a negative 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.[8] 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 $\{1, 2, \ldots, \nu\}$), and assign to each *variable* x_n a weight drawn u.a.r. from

- $\mathcal{U}\{0, 1\}$ if $\pi(n) \leq \nu\delta$,
- $\mathcal{U}\{0.5\}$ if $\nu\delta < \pi(n) \leq \nu\delta + \nu\epsilon$,
- and $\mathcal{U}\{0.01, 0.02, \ldots, 0.99\}$[9] 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 $\{x_1, x_2, \ldots, x_5\}$. 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 $N = \varnothing$. 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., $\neg x_5 \vee x_2 \vee x_1$.

The first variable of the second clause is chosen u.a.r. from $\{x_1, x_2, \ldots, x_5\}$. 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

[8] Note that algorithms such as DPMC and ADDMC [34, 35] support a more flexible way of assigning weights that can lead to significant performance improvements [30, 31].

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

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 $\{0.01, 0.02, \ldots, 0.99\}$. The weight function $w: \{x_1, x_2, \ldots, x_5, \neg x_1, \neg x_2, \ldots, \neg x_5\} \to [0, 1]$ 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. We fix $\nu = 100, \delta = \epsilon = 0$, and consider random instances with $\mu = 2.5 \times \sqrt{2}^{-5}, 2.5 \times \sqrt{2}^{-4}, \ldots, 2.5 \times \sqrt{2}^{5}$, $\kappa = 2, 3, 4, 5$, and ρ going from 0 to 1 in steps of 0.01. For each combination of parameters, we generate ten instances.[10] We check if each instance is satisfiable using MiniSat[11] 2.2.0 [37] and calculate its (approximate) primal treewidth using htd[12] [1].

Remark 1. Here and henceforth, we use htd to provide heuristic upper bounds on true treewidth as exact computation would make the experiments significantly more time-consuming. However, we compared the accuracy of htd with another treewidth algorithm Jdrasil[13] [6] (set to compute exact values) on 3 % of our random instances. The difference between the upper bound produced by htd and the exact value was never higher than 4 and at most 2 in 85 % of all cases.

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

[10] Since one expects similar values of ρ to produce instances with similar properties, and ρ's are enumerate quite densely, generating only ten instances is sufficient.
[11] http://minisat.se/MiniSat.html
[12] https://github.com/mabseher/htd
[13] https://maxbannach.github.io/Jdrasil/

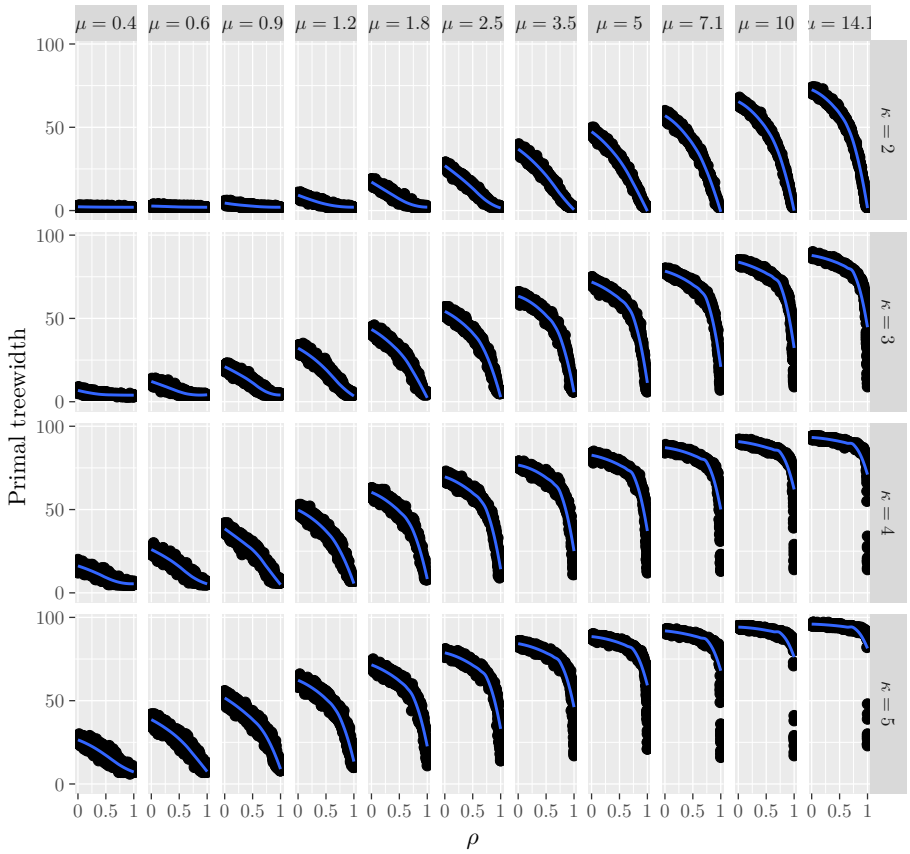

Fig. 1. The relationship between ρ and primal treewidth for various values of μ and κ for k-CNF formulas from Experiment 1. Black points represent individual instances, and blue lines are smoothed means computed using locally weighted smoothing. The values of μ are rounded to one decimal place.

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 [20].

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 setup in the next section.

5 Experimental Results

In Section 5.1, we examine how the running times of WMC algorithms change w.r.t. the parameters of our random model. Then, in Section 5.2, we run an

experiment with WMC competition benchmarks to check whether the conclusions drawn from random instances apply to real data.

For all of these experiments, we use Scientific Linux 7, GCC 10.2.0, Python 3.8.1, R 4.1.0, C2D 2.20 [26], CACHET 1.22 [63], HTD 1.2.0 [1], and perform no preprocessing. With both C2D and D4 [52], we use QUERY-DNNF[14] to compute the numerical answer from the compiled circuit. We omit ADDMC [34] from our experiments as it exceeds time and memory limits on too many instances; however, observations about the behaviour of DPMC [35] 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 (using one iteration of HTD) and ADD-based execution—the combination that was originally found to be most effective.

5.1 Experiments on Random Instances

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 on Intel Xeon E5–2630. 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 $\nu = 70$, $\mu = 2.2$[15], $\rho = 0$, 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 [52] and C2D 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 Figure 2. The first thing to note is that the peak hardness w.r.t. density occurs at around 1.9 for all algorithms except for DPMC, which peaks at

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

[15] Experiment 2 shows this density to be the most challenging for DPMC.

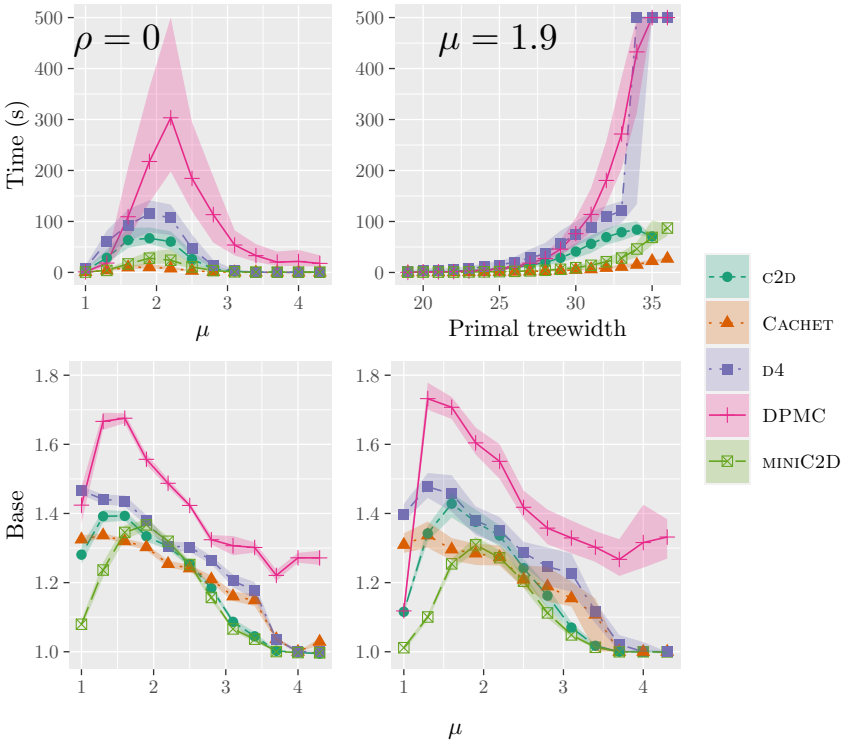

Fig. 2. Visualisations of the data from Experiment 2. The top-left plot shows how the running time of each algorithm changes w.r.t. density when $\rho = 0$. The top-right plot shows changes in the running time of each algorithm w.r.t. primal treewidth with μ fixed at 1.9. The plots at the bottom show how the estimated base of the exponential relationship between primal treewidth and the runtime of each algorithm depends on μ. The bottom-left plot is for the simple linear model (with shaded regions showing standard error), and the bottom-right plot uses the estimates provided by ESA [58] (with shaded regions showing 95 % confidence intervals).

2.2 instead.[16] This finding is consistent with previous work, which shows Cachet to peak at 1.8 [63].[17]

The other question we want to investigate using this experiment is how each algorithm scales w.r.t. primal treewidth. The top-right plot in Figure 2 shows this relationship for a fixed value of μ, and one can see some evidence that the running time of DPMC grows faster w.r.t. primal treewidth than the running time of the other algorithms. We use two statistical techniques to quantify this growth: a

[16] While exact values might be hard to read from the plot, they are confirmed by numerical data.

[17] For comparison, #SAT algorithms have been observed to peak at densities 1.2 and 1.5 [8].

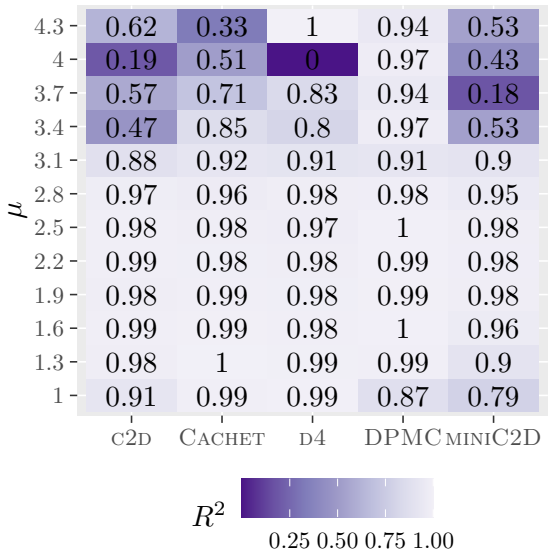

Fig. 3. The coefficients of determination (rounded to one decimal place) of all the linear models fitted for the top-right subplot of Figure 2

simple linear regression model and the empirical scaling analyzer (ESA) v2[18] [58]. In both cases, for each algorithm and value of μ in Experiment 2, we select the median runtime for all available values of primal treewidth. In the former case, we 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.[19] In other words, this model attempts to express median running time as $e^{\beta}(e^{\alpha})^{w}$. In the latter case, we run ESA with 1001 bootstrap samples, a window of 101, and use the first 30 % of the data for training.

The results of both models are qualitatively the same (with the exception of DPMC run on instances with $\mu = 1$) and are displayed at the bottom of Figure 2. 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 Figure 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 Figure 3 shows that the fitted models fail to explain the data for non-ADD algorithms running

[18] https://github.com/YashaPushak/ESA

[19] Similar statistical analyses have been used to investigate polynomial-to-exponential phase transitions in SAT [18] and the behaviour of SAT solvers on CNF-XOR formulas [33].

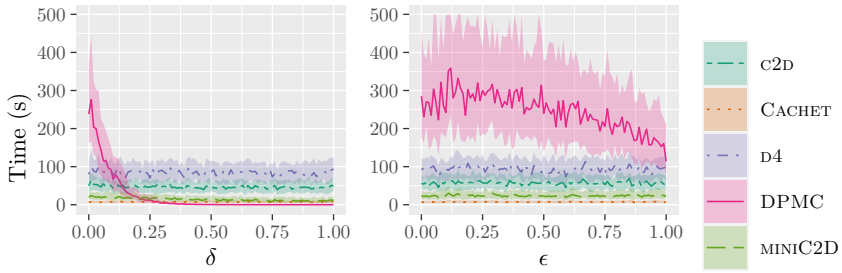

Fig. 4. Changes in the running time of each algorithm as a result of changing δ (on the left-hand side) and ϵ (on the right-hand side) according to the data from Experiments 3 and 4

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 [55] is the only algorithm that exhibits a clear low-high-low pattern in the bottom subplots of Figure 2. To a smaller extent, the same may apply to c2d and DPMC as well, although 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 and d4 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.

Finally, Experiments 3 and 4 investigate how changing the numerical values of weights can simplify a WMC instance. The results are plotted in Figure 4. As expected, the running time of all algorithms other than DPMC 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 et al. [65] as found in the experimental setup[20] of the DPMC paper [35]. A typical 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

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

of all other algorithms on traditionally-used benchmarks [35] despite struggling with most of our random data.

To sum, we found that C2D and D4 are the most memory-intensive algorithms, CACHET is great on random instances in general, MINIC2D exceeds on low-density high-primal-treewidth instances, and DPMC 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.

5.2 Experiments on Competition Benchmarks

To check whether our observations on random instances are accurate on real data, we use the 100 public instances from track 2 of the 2022 model counting competition[21]. This time, the WMC algorithms are run on Intel Xeon Gold 6138 with 32 GiB of memory and a 1 h time limit. As in Section 5.1, we compute the density and the primal treewidth of each instance.

Figure 5 shows the best-performing algorithm for various combinations of the parameters. We observe that:

- DPMC [35] is best on most instances with low primal treewidth,
- C2D [26] is able to handle some low-density high-primal-treewidth instances that all the other algorithms fail on,
- CACHET [63] (as well as D4 [52] to some extent) excel when both density and primal treewidth are quite high,
- and MINIC2D [55] does not have an easily-recognisable niche.

TODO: compare these results with the previous section.

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 [36] as well as to other applications of ADDs more generally, e.g., probabilistic inference [15, 45] and stochastic planning [48].

[21] https://mccompetition.org/2022/mc_description

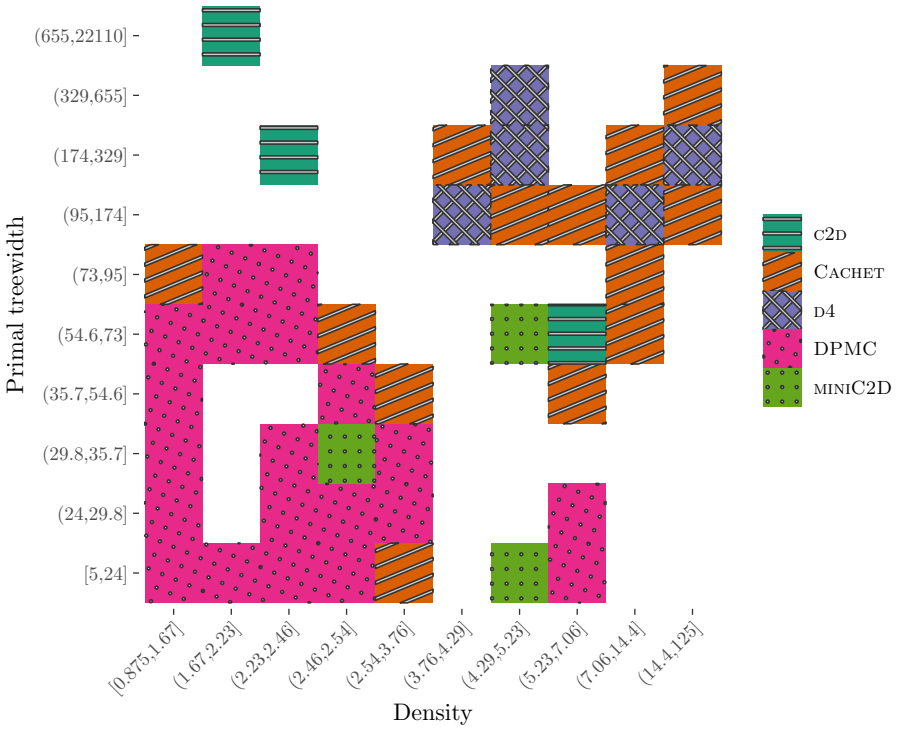

Fig. 5. The best-performing algorithm for each combination of density and primal treewidth according to the experiments on competition benchmarks. Both ranges of values are divided into ten bins so that there are ten instances in each bin. The best-performing algorithm for each combination of bins is the algorithm that solved the largest number of instances, with ties broken by minimising total running time. An empty cell means that either no benchmark had this combination of density and primal treewidth or all algorithms failed on all such instances.

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 [32], 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 [25].

References

1. Abseher, M., Musliu, N., Woltran, S.: htd - A free, open-source framework for (customized) tree decompositions and beyond. In: CPAIOR. Lecture Notes in Computer Science, vol. 10335, pp. 376–386. Springer (2017)
2. Achlioptas, D., Moore, C.: The asymptotic order of the random k -sat threshold. In: FOCS. pp. 779–788. IEEE Computer Society (2002)
3. Ansótegui, C., Bonet, M.L., Levy, J.: Towards industrial-like random SAT instances. In: IJCAI. pp. 387–392 (2009)
4. Bacchus, F., Dalmao, S., Pitassi, T.: Solving #SAT and Bayesian inference with backtracking search. J. Artif. Intell. Res. **34**, 391–442 (2009)
5. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Formal Methods Syst. Des. **10**(2/3), 171–206 (1997)
6. Bannach, M., Berndt, S., Ehlers, T.: Jdrasil: A modular library for computing tree decompositions. In: SEA. LIPIcs, vol. 75, pp. 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
7. Bart, A., Koriche, F., Lagniez, J., Marquis, P.: An improved CNF encoding scheme for probabilistic inference. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 613–621. IOS Press (2016)
8. Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: AAAI/IAAI. pp. 157–162. AAAI Press / The MIT Press (2000)
9. Belle, V.: Open-universe weighted model counting. In: AAAI. pp. 3701–3708. AAAI Press (2017)
10. Belle, V., Passerini, A., Van den Broeck, G.: Probabilistic inference in hybrid domains by weighted model integration. In: IJCAI. pp. 2770–2776. AAAI Press (2015)
11. Bläsius, T., Friedrich, T., Sutton, A.M.: On the empirical time complexity of scale-free 3-SAT at the phase transition. In: TACAS (1). Lecture Notes in Computer Science, vol. 11427, pp. 117–134. Springer (2019)
12. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI. pp. 689–695. AAAI Press (2015)
13. Chavira, M., Darwiche, A.: Compiling bayesian networks with local structure. In: IJCAI. pp. 1306–1312. Professional Book Center (2005)
14. Chavira, M., Darwiche, A.: Encoding cnfs to empower component analysis. In: SAT. Lecture Notes in Computer Science, vol. 4121, pp. 61–74. Springer (2006)
15. Chavira, M., Darwiche, A.: Compiling bayesian networks using variable elimination. In: IJCAI. pp. 2443–2449 (2007)
16. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6-7), 772–799 (2008)
17. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational bayesian networks for exact inference. Int. J. Approx. Reason. **42**(1-2), 4–20 (2006)
18. Coarfa, C., Demopoulos, D.D., Aguirre, A.S.M., Subramanian, D., Vardi, M.Y.: Random 3-SAT: The plot thickens. Constraints An Int. J. **8**(3), 243–261 (2003)
19. Coja-Oghlan, A., Wormald, N.: The number of satisfying assignments of random regular k-SAT formulas. Comb. Probab. Comput. **27**(4), 496–530 (2018)

20. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in random 3-SAT. Artif. Intell. **81**(1-2), 31–57 (1996)
21. Dal, G.H., Laarman, A.W., Lucas, P.J.F.: Parallel probabilistic inference by weighted model counting. In: PGM. Proceedings of Machine Learning Research, vol. 72, pp. 97–108. PMLR (2018)
22. Darwiche, A.: Compiling knowledge into decomposable negation normal form. In: IJCAI. pp. 284–289. Morgan Kaufmann (1999)
23. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
24. Darwiche, A.: 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 (2001)
25. Darwiche, A.: A logical approach to factoring belief networks. In: KR. pp. 409–420. Morgan Kaufmann (2002)
26. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: ECAI. pp. 328–332. IOS Press (2004)
27. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: IJCAI. pp. 819–826. IJCAI/AAAI (2011)
28. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**, 229–264 (2002)
29. Dilkas, P., Belle, V.: Generating random logic programs using constraint programming. In: CP. Lecture Notes in Computer Science, vol. 12333, pp. 828–845. Springer (2020)
30. Dilkas, P., Belle, V.: Weighted model counting with conditional weights for Bayesian networks. In: UAI. Proceedings of Machine Learning Research, vol. 161, pp. 386–396. AUAI Press (2021)
31. Dilkas, P., Belle, V.: Weighted model counting without parameter variables. In: SAT. Lecture Notes in Computer Science, vol. 12831, pp. 134–151. Springer (2021)
32. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Texts in Computer Science, Springer (2013)
33. Dudek, J.M., Meel, K.S., Vardi, M.Y.: The hard problems are almost everywhere for random CNF-XOR formulas. In: IJCAI. pp. 600–606. ijcai.org (2017)
34. Dudek, J.M., Phan, V., Vardi, M.Y.: ADDMC: weighted model counting with algebraic decision diagrams. In: AAAI. pp. 1468–1476. AAAI Press (2020)
35. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: DPMC: weighted model counting by dynamic programming on project-join trees. In: CP. Lecture Notes in Computer Science, vol. 12333, pp. 211–230. Springer (2020)
36. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: Procount: Weighted projected model counting with graded project-join trees. In: SAT. Lecture Notes in Computer Science, vol. 12831, pp. 152–170. Springer (2021)
37. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
38. Fargier, H., Marquis, P.: On the use of partially ordered decision graphs in knowledge compilation and quantified boolean formulae. In: AAAI. pp. 42–47. AAAI Press (2006)
39. Fichte, J.K., Hecher, M., Woltran, S., Zisser, M.: Weighted model counting on the GPU by exploiting small treewidth. In: ESA. LIPIcs, vol. 112, pp. 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
40. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Program. **15**(3), 358–401 (2015)

41. Franco, J., Paull, M.C.: Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. Discret. Appl. Math. **5**(1), 77–87 (1983)
42. Galanis, A., Goldberg, L.A., Guo, H., Yang, K.: Counting solutions to random CNF formulas. In: ICALP. LIPIcs, vol. 168, pp. 53:1–53:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
43. Giráldez-Cru, J., Levy, J.: Generating SAT instances with community structure. Artif. Intell. **238**, 119–134 (2016)
44. Giráldez-Cru, J., Levy, J.: Locality in random SAT instances. In: IJCAI. pp. 638–644. ijcai.org (2017)
45. Gogate, V., Domingos, P.M.: Approximation by quantization. In: UAI. pp. 247–255. AUAI Press (2011)
46. Gogate, V., Domingos, P.M.: Probabilistic theorem proving. Commun. ACM **59**(7), 107–115 (2016)
47. Hecher, M., Thier, P., Woltran, S.: Taming high treewidth with abstraction, nested dynamic programming, and database technology. In: SAT. Lecture Notes in Computer Science, vol. 12178, pp. 343–360. Springer (2020)
48. Hoey, J., St-Aubin, R., Hu, A.J., Boutilier, C.: SPUDD: stochastic planning using decision diagrams. In: UAI. pp. 279–288. Morgan Kaufmann (1999)
49. Holtzen, S., Van den Broeck, G., Millstein, T.D.: Scaling exact inference for discrete probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA), 140:1–140:31 (2020)
50. Hossain, M.M., Abbass, H.A., Lokan, C., Alam, S.: Adversarial evolution: Phase transition in non-uniform hard satisfiability problems. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2010)
51. Korhonen, T., Järvisalo, M.: Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In: CP. LIPIcs, vol. 210, pp. 8:1–8:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
52. Lagniez, J., Marquis, P.: An improved decision-dnnf compiler. In: IJCAI. pp. 667–673. ijcai.org (2017)
53. Mitchell, D.G., Selman, B., Levesque, H.J.: Hard and easy distributions of SAT problems. In: AAAI. pp. 459–465. AAAI Press / The MIT Press (1992)
54. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535. ACM (2001)
55. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: IJCAI. pp. 3141–3148. AAAI Press (2015)
56. Pote, Y., Joshi, S., Meel, K.S.: Phase transition behavior of cardinality and XOR constraints. In: IJCAI. pp. 1162–1168. ijcai.org (2019)
57. Purdom Jr., P.W., Brown, C.A.: An analysis of backtracking with search rearrangement. SIAM J. Comput. **12**(4), 717–733 (1983)
58. Pushak, Y., Hoos, H.H.: Advanced statistical analysis of empirical performance scaling. In: GECCO. pp. 236–244. ACM (2020)
59. Renkens, J., Kimmig, A., Van den Broeck, G., De Raedt, L.: Explanation-based approximate weighted model counting for probabilistic logics. In: AAAI. pp. 2490–2496. AAAI Press (2014)
60. Riguzzi, F.: Quantum weighted model counting. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2640–2647. IOS Press (2020)
61. Robertson, N., Seymour, P.D.: Graph minors. III. planar tree-width. J. Comb. Theory, Ser. B **36**(1), 49–64 (1984)
62. Robertson, N., Seymour, P.D.: Graph minors. x. obstructions to tree-decomposition. J. Comb. Theory, Ser. B **52**(2), 153–190 (1991)

63. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT (2004)
64. Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: SAT. Lecture Notes in Computer Science, vol. 3569, pp. 226–240. Springer (2005)
65. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI. pp. 475–482. AAAI Press / The MIT Press (2005)
66. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A scalable probabilistic exact model counter. In: IJCAI. pp. 1169–1176. ijcai.org (2019)
67. Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., De Raedt, L.: Lifted probabilistic inference by first-order knowledge compilation. In: IJCAI. pp. 2178–2185. IJCAI/AAAI (2011)
68. Xu, J., Zhang, Z., Friedman, T., Liang, Y., Van den Broeck, G.: A semantic loss function for deep learning with symbolic knowledge. In: ICML. Proceedings of Machine Learning Research, vol. 80, pp. 5498–5507. PMLR (2018)
69. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)