Handbook of Satisfiability
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch
IOS Press, 2008

Chapter 20

Model Counting

Carla P. Gomes, Ashish Sabharwal, and Bart Selman

Propositional model counting or #SAT is the problem of computing the number of models for a given propositional formula, i.e., the number of distinct truth assignments to variables for which the formula evaluates to TRUE. For a propositional formula F, we will use $\#F$ to denote the model count of F. This problem is also referred to as the solution counting problem for SAT. It generalizes SAT and is the canonical #P-complete problem. There has been significant theoretical work trying to characterize the worst-case complexity of counting problems, with some surprising results such as model counting being hard even for some polynomial-time solvable problems like 2-SAT.

The model counting problem presents fascinating challenges for practitioners and poses several new research questions. Efficient algorithms for this problem will have a significant impact on many application areas that are inherently beyond SAT ('beyond' under standard complexity theoretic assumptions), such as bounded-length adversarial and contingency planning, and probabilistic reasoning. For example, various probabilistic inference problems, such as Bayesian net reasoning, can be effectively translated into model counting problems [cf. 2, 11, 30, 38, 41, 45]. Another application is in the study of hard combinatorial problems, such as combinatorial designs, where the number of solutions provides further insights into the problem. Even finding a single solution can be a challenge for such problems; counting the number of solutions is much harder. Not surprisingly, the largest formulas we can solve for the model counting problem with state-of-the-art model counters are orders of magnitude smaller than the formulas we can solve with the best SAT solvers. Generally speaking, current exact counting methods can tackle problems with a couple of hundred variables, while approximate counting methods push this to around 1,000 variables.

#SAT can be solved, in principle and to an extent in practice, by extending the two most successful frameworks for SAT algorithms, namely, DPLL and local search. However, there are some interesting issues and choices that arise when extending SAT-based techniques to this harder problem. In general, solving #SAT requires the solver to, in a sense, be cognizant of all solutions in the search space, thereby reducing the effectiveness and relevance of commonly used SAT heuristics designed to quickly narrow down the search to a single solution. The resulting

scalability challenge has drawn many satisfiability researchers to this problem, and to the related problem of sampling solutions uniformly at random.

We will divide practical model counting techniques we consider into two main categories: *exact counting* and *approximate counting*, discussed in Sections 20.2 and 20.3, respectively. Within exact counting, we will distinguish between methods based on *DPLL-style* exhaustive search (Section 20.2.1) and those based on "knowledge compilation" or conversion of the formula into certain *normal forms* (Section 20.2.2). Within approximate counting, we will distinguish between methods that provide fast *estimates without any guarantees* (Section 20.3.1) and methods that provide lower or upper *bounds with a correctness guarantee*, often in a probabilistic sense and recently also in a statistical sense (Section 20.3.2).

We would like to note that there are several other directions of research related to model counting that we will not cover here. For example, Nishimura et al. [36] explore the concept of "backdoors" for #SAT, and show how the vertex cover problem can be used to identify small such backdoors based on so-called cluster formulas. Bacchus et al. [2] consider structural restrictions on the formula and propose an algorithm for #SAT whose complexity is polynomial in n (the number of variables) and exponential in the "branch-width" of the underlying constraint graph. Gottlob et al. [23] provide a similar result in terms of "tree-width". Fischer et al. [15] extend this to a similar result in terms of "cluster-width" (which is never more than tree-width, and sometimes smaller). There is also complexity theoretic work on this problem by the theoretical computer science community. While we do provide a flavor of this work (Section 20.1), our focus will mostly be on techniques that are available in the form of implemented and tested model counters.

20.1. Computational Complexity of Model Counting

We begin with a relatively brief discussion of the theoretical foundations of the model counting problem. The reader is referred to standard complexity texts [cf. 37] for a more detailed treatment of the subject.

Given any problem in the class NP, say SAT or CLIQUE, one can formulate the corresponding *counting problem*, asking *how many solutions exist* for a given instance? More formally, given a polynomial-time decidable relation Q,[1] the corresponding counting problem asks: given x as input, how many y's are there such that $(x, y) \in Q$? For example, if Q is the relation "y is a truth assignment that satisfies the propositional expression x" then the counting problem for Q is the propositional model counting problem, #SAT. Similarly, if Q is the relation "y is a clique in the graph x" then the counting problem for Q is #CLIQUE. The complexity class #P (pronounced "number P" or "sharp P") consists of all counting problems associated with such polynomial-time decidable relations. Note that the corresponding problem in NP asks: given x, *does there exist* a y such that $(x, y) \in Q$?

The notion of *completeness* for #P is defined essentially in the usual way, with a slight difference in the kind of reduction used. A problem A is #P-

[1] Technically, Q must also be polynomially balanced, that is, for each x, the only possible y's with $(x, y) \in Q$ satisfy $|y| \leq |x|^k$ for a constant k.

complete if (1) A is in #P, and (2) for every problem B in #P, there exists a polynomial-time *counting reduction* from B to A. A counting reduction is an extension of the reductions one often encounters between NP-complete decision problems, and applies to function computation problems. There are two parts to a counting reduction from B to A: a polynomial-time computable function R that maps an instance z of B to an instance $R(z)$ of A, and a polynomial-time computable function S that recovers from the count N of $R(z)$ the count $S(N)$ of z. In effect, given an algorithm for the counting problem A, the relations R and S together give us a recipe for converting that algorithm into one for the counting problem B with only a polynomial overhead.

Conveniently, many of the known reductions between NP-complete problems are already *parsimonious*, that is, they preserve the number of solutions during the translation. Therefore, these reductions can be directly taken to be the R part of a counting reduction, with the trivial identity function serving as the S part, thus providing an easy path to proving #P-completeness. In fact, one can construct a parsimonious version of the Cook-Levin construction, thereby showing that #SAT is a canonical #P-complete problem. As it turns out, the solution counting variants of all six basic NP-complete problems listed by Garey and Johnson [16], and of many more NP-complete problems, are known to be #P-complete.[2]

In his seminal paper, Valiant [51] proved that, quite surprisingly, the solution counting variants of polynomial-time solvable problems can also be #P-complete. Such problems, in the class P, include 2-SAT, Horn-SAT, DNF-SAT, bipartite matching, etc. What Valiant showed is that the problem PERM of computing the *permanent* of a 0-1 matrix, which is equivalent to counting the number of perfect matchings in a bipartite graph or #BIP-MATCHING, is #P-complete. On the other hand, the corresponding search problem of computing a *single* perfect matching in a bipartite graph can be solved in deterministic polynomial time using, e.g., a network flow algorithm. Therefore, unless P=NP, there does not exist a *parsimonious* reduction from SAT to BIP-MATCHING; such a reduction would allow one to solve any SAT instance in polynomial time by translating it to a BIP-MATCHING instance and checking for the existence of a perfect matching in the corresponding bipartite graph. Valiant instead argued that there is a smart way to *indirectly* recover the answer to a #SAT instance from the answer to the corresponding PERM (or #BIP-MATCHING) instance, using a non-identity polynomial-time function S in the above notation of counting reductions.

Putting counting problems in the traditional complexity hierarchy of decision problems, Toda [49] showed that #SAT being #P-complete implies that it is no easier than solving a quantified Boolean formula (QBF) with a constant number (independent of n, the number of variables) of "there exist" and "forall" quantification levels in its variables. For a discussion of the QBF problem, see Part 2, Chapters 23-24 of this Handbook. Formally, Toda considered the decision problem class $\mathrm{P}^{\#\mathrm{P}}$ consisting of polynomial-time decision computations with "free" access to #P queries, and compared this with k-QBF, the subset of QBF instances that have exactly k quantifier alternations, and the infinite polynomial

[2] Note that a problem being NP-complete does not automatically make its solution counting variant #P-complete; one must demonstrate a polynomial time *counting* reduction.

hierarchy $\text{PH} = \bigcup_{k=1}^{\infty} k\text{-QBF}$.[3] Combining his result with the relatively easy fact that counting problems can be solved in polynomial space, we have #P placed as follows in the worst-case complexity hierarchy:

$$\text{P} \subseteq \text{NP} \subseteq \text{PH} \subseteq \text{P}^{\#\text{P}} \subseteq \text{PSPACE}$$

where PSPACE is the class of problems solvable in polynomial space, with QBF being the canonical PSPACE-complete problem. As a comparison, notice that SAT can be thought of as a QBF with exactly one level of "there exist" quantification for all its variables, and is thus a subset of 1-QBF. While the best known deterministic algorithms for SAT, #SAT, and QBF all run in worst-case exponential time, it is widely believed—by theoreticians and practitioners alike—that #SAT and QBF are significantly harder to solve than SAT.

While #P is a class of function problems (rather than decision problems, for which the usual complexity classes like P and NP are defined), there does exist a natural variant of it called PP (for "probabilistic polynomial time") which is a class of essentially equally hard decision problems. For a polynomial-time decidable relation Q, the corresponding PP problem asks: given x, is $(x, y) \in Q$ for *more than half* the y's? The class PP is known to contain both NP and co-NP, and is quite powerful. The proof of Toda's theorem mentioned earlier in fact relies on the equality $\text{P}^{\text{PP}} = \text{P}^{\#\text{P}}$ observed by Angluin [1]. One can clearly answer the PP query for a problem given the answer to the corresponding #P query. The other direction is a little less obvious, and uses the fact that PP has the power to provide the "most significant bit" of the answer to the #P query and it is possible to obtain all bits of this answer by repeatedly querying the PP oracle on appropriately modified problem instances.

We close this section with a note that Karp and Luby [25] gave a Markov Chain Monte Carlo (MCMC) search based fully polynomial-time randomized approximation scheme (FPRAS) for the DNF-SAT model counting problem, and Karp et al. [26] later improved its running time to yield the following: for any $\epsilon, \delta \in (0, 1)$, there exists a randomized algorithm that given F computes an ϵ-approximation to #F with correctness probability $1 - \delta$ in time $O(|F| \cdot 1/\epsilon^2 \cdot \ln(1/\delta))$, where $|F|$ denotes the size of F.

20.2. Exact Model Counting

We now move on to a discussion of some of the practical implementations of exact model counting methods which, upon termination, output the true model count of the input formula. The "model counters" we consider are `CDP` by Birnbaum and Lozinskii [7], `Relsat` by Bayardo Jr. and Pehoushek [4], `Cachet` by Sang et al. [43], `sharpSAT` by Thurley [48], and `c2d` by Darwiche [10].

[3] For simplicity, we use k-QBF to represent not only a subset of QBF formulas but also the complexity class of alternating Turing machines with k alternations, for which solving these formulas forms the canonical complete problem.

20.2.1. DPLL-Based Model Counters

Not surprisingly, the earliest practical approach for counting models is based on an extension of systematic DPLL-style SAT solvers. The idea, formalized early by Birnbaum and Lozinskii [7] in their model counter `CDP`, is to directly explore the complete search tree for an n-variable formula as in the usual DPLL search, pruning unsatisfiable branches based on falsified clauses and declaring a branch to be satisfied when all clauses have at least one TRUE literal. However, unlike the usual DPLL, when a branch is declared satisfied and the partial truth assignment at that point has t fixed variables (fixed either through the branching heuristic or by unit propagation), we associate 2^{n-t} solutions with this branch corresponding to the partial assignment being extended by all possible settings of the $n-t$ yet unset variables, backtrack to the last decision variable that can be flipped, flip that variable, and continue exploring the remaining search space. The model count for the formula is finally computed as the sum of such 2^{n-t} counts obtained over all satisfied branches. Although all practical implementations of DPLL have an iterative form, it is illustrative to consider `CDP` in a recursive manner, written here as Algorithm 20.1, where $\#F$ is computed as the sum of $\#F|_x$ and $\#F|_{\neg x}$ for a branch variable x, with the discussion above reflected in the two base cases of this recursion.

Algorithm 20.1: `CDP` $(F, 0)$

Input : A CNF formula F over n variables; a parameter t initially set to 0
Output : $\#F$, the model count of F
begin
  `UnitPropagate(`F`)`
  if *F has an empty clause* **then return** 0
  if *all clauses of F are satisfied* **then return** 2^{n-t}
  $x \leftarrow$ `SelectBranchVariable(`F`)`
  return `CDP(`$F|_x, t+1$`)` + `CDP(`$F|_{\neg x}, t+1$`)`
end

An interesting technical detail is that many of the modern implementations of DPLL do *not* maintain data structures that would allow one to easily check whether or not all clauses have been satisfied by the current partial assignment. In general, DPLL-style SAT solvers often do not explicitly keep track of the number of unsatisfied clauses. They only keep track of the number of assigned variables, and declare success when all variables have been assigned values and no clause is violated. Keeping track of the number of unsatisfied clauses is considered unnecessary because once a partial assignment happens to satisfy all clauses, further branching immediately sets all remaining variables to arbitrary values and obtains a complete satisfying assignment; complete satisfying assignments are indeed what many applications of SAT seek. DPLL-based model counters, on the other hand, do maintain this added information about how many clauses are currently satisfied and infer the corresponding 2^{n-t} counts. Having to enumerate each of the 2^{n-t} complete satisfying assignments instead would make the technique impractical.

Obtaining partial counts: As discussed above, a basic DPLL-based model counter works by using appropriate multiplication factors and continuing the search after a single solution is found. An advantage of this approach is that the model count is computed in an incremental fashion: if the algorithm runs out of a pre-specified time limit, it can still output a correct *lower bound* on the true model count, based on the part of the search space it has already explored. This can be useful in many applications, and has been the motivation for some new randomized techniques that provide fast lower bounds with probabilistic correctness guarantees (to be discussed in Section 20.3.2). In fact, a DPLL-based model counter can, in principle, also output a correct *upper bound* at any time: 2^n minus the sum of the 2^{n-t} style counts of un-satisfying assignments contained in all unsatisfiable branches explored till that point. Unfortunately, this is often not very useful in practice because the number of solutions of problems of interest is typically much smaller than the size of the search space. As a simple example, in a formula with 1000 variables and 2^{200} solutions, after having explored, say, a 1/16 fraction of the search space, it is reasonable to expect the model counter to have seen roughly $2^{200}/16 = 2^{196}$ solutions (which would be a fairly good lower bound on the model count) while one would expect to have seen roughly $(2^{1000} - 2^{200})/16$ un-satisfying assignments (yielding a poor naïve upper bound of $2^{1000} - (2^{1000} - 2^{200})/16$, which is at least as large as 2^{999}). We will discuss a more promising, statistical upper bounding technique towards the end of this chapter.

Component analysis: Consider the constraint graph G of a CNF formula F. The vertices of G are the variables of F and there is an edge between two vertices if the corresponding variables appear together in some clause of F. Suppose G can be partitioned into disjoint components $G_1, G_2, \ldots, G_k$ where there is no edge connecting a vertex in one component to a vertex in another component, i.e., the variables of F corresponding to vertices in two different components do not appear together in any clause. Let $F_1, F_2, \ldots, F_k$ be the sub-formulas of F corresponding to the k components of G and restricted to the variables that appear in the corresponding component. Since the components are disjoint, it follows that every clause of F appears in a unique component, the sub-problems captured by the components are independent, and, most pertinent to this chapter, that $\#F = \#F_1 \times \#F_2 \times \ldots \times \#F_k$. Thus, $\#F$ can be evaluated by identifying disjoint components of F, computing the model count of each component, and multiplying the results together.

This idea is implemented in one of the first effective exact model counters for SAT, called `Relsat` [4], which extends a previously introduced DPLL-based SAT solver by the same name [5]. Components are identified dynamically as the underlying DPLL procedure attempts to extend a partial assignment. With each new extension, several clauses may be satisfied so that the constraint graph simplifies dynamically depending on the actual value assignment to variables. While such dynamic detection and exploitation of components has often been observed to be too costly for pure satisfiability testing,[4] it certainly pays off well for the harder task of model counting. Note that for the correctness of the

[4] Only recently have SAT solvers begun to efficiently exploit partial component caching schemes [40].

method, all we need is that the components are disjoint. However, the components detected by `Relsat` are, in fact, the *connected* components of the constraint graph of F, indicating that the full power of this technique is being utilized. One of the heuristic *optimizations* used in `Relsat` is to attempt the most constrained sub-problems first. Another trick is to first check the satisfiability of every component, before attempting to count any. `Relsat` also solves sub-problems in an interleaved fashion, dynamically jumping to another sub-problem if the current one turns out to be less constrained than initially estimated, resulting in a best-first search of the developing component tree. Finally, the component structure of the formula is determined lazily while backtracking, instead of eagerly before branch selection. This does not affect the search space explored but often reduces the component detection overhead for unsatisfiable branches.

Bayardo Jr. and Schrag [5] demonstrated through `Relsat` that applying these ideas significantly improves performance over basic `CDP`, obtaining exact counts for several problems from graph coloring, planning, and circuit synthesis/analysis domains that could not be counted earlier. They also observed, quite interestingly, that the *peak of hardness* of model counting for random 3-SAT instances occurs at a very different clause-to-variable ratio than the peak of hardness of solving such formulas. These instances were found to be the hardest for model counting at a ratio of $\alpha \approx 1.5$, compared to $\alpha \approx 4.26$ which marks the peak of hardness for SAT solvers as well as the (empirical) point of phase transition in such formulas from being mostly satisfiable to mostly unsatisfiable. Bailey et al. [3] followed up on this observation and provided further analytical and empirical results on the hardness peak and the corresponding phase transition of the decision variants of random counting problems.

Caching: As one descends the search tree of a DPLL-based model counter, setting variables and simplifying the formula, one may encounter sub-formulas that have appeared in an earlier branch of the search tree. If this happens, it would clearly be beneficial to be able to efficiently recognize this fact, and instead of re-computing the model count of the sub-formula from scratch, somehow "remember" the count computed for it earlier. This is, in principle, similar to the clause learning techniques used commonly in today's SAT solvers, except that for the purposes of model counting, it is no longer possible to succinctly express the key knowledge learned from each previous sub-formula as a single "conflict clause" that, for SAT, quite effectively captures the "reason" for that sub-formula being unsatisfiable. For model counting, one must also store, in some form, a signature of the full satisfiable sub-formulas encountered earlier, along with their computed model counts. This is the essence of *formula caching* systems [2, 6, 32]. While formula caching is theoretically appealing even for SAT, being more powerful than clause learning [6], its overhead is much more likely to be offset when applied to harder problems like #SAT.

Bacchus et al. [2] considered three variants of caching schemes: simple caching (a.k.a. formula caching), component caching, and linear-space caching. They showed that of these, *component caching* holds the greatest promise, being theoretically competitive with (and sometimes substantially better than) some of the best known methods for Bayesian inference. Putting these ideas into practice,

Sang et al. [43] created the model counter `Cachet`, which ingeniously combined component caching with traditional clause learning within the setup of model counting.[5] `Cachet` is built upon the well-known SAT solver `zChaff` [34]. It turns out that combining component caching and clause learning in a naïve way leads to subtle issues that would normally permit one to only compute a lower bound on the model count. This problem is taken care of in `Cachet` using so-called *sibling pruning*, which prevents the undesirable interaction between cached components and clause learning from spreading.

Taking these ideas further, Sang et al. [44] considered the efficiency of various heuristics used in SAT solvers, but now in the context of model counting with component caching. They looked at component selection strategies, variable selection branching heuristics, randomization, backtracking schemes, and cross-component implications. In particular, they showed that model counting works better with a variant of the conflict graph based branching heuristic employed by `zChaff`, namely VSIDS (variable state independent decaying sum). This variant is termed VSADS, for variable state *aware* decaying sum, which linearly interpolates between the original VSIDS score and a more traditional formula-dependent score based on the number of occurrences of each variable.

Improved caching and more reasoning at each node: An important concern in implementing formula caching or component caching in practice is the space requirement. While these concerns are already present even for clause learning techniques employed routinely by SAT solvers and have led to the development of periodic clause deletion mechanisms, the problem is clearly more severe when complete sub-formulas are cached. `sharpSAT` [48] uses several ideas that let components be stored more succinctly. For example, all clauses of any component stored by `sharpSAT` have at least two unassigned literals (unit propagation takes care of any active clauses with only one unassigned literal), and it does not explicitly store any binary clauses of the original formula in the component signature (binary clauses belonging to the component have both literals unassigned and can thus be easily reconstructed from the set of variables associated with the component). Further, it only stores (1) the indices of the variables in the component and (2) the indices of the original clauses that belong to that component, rather than storing full clauses or the learned conflict clauses. This can, in principle, prohibit some components from being identified as identical when they would be identified as identical by `Cachet`, which stores full clauses. Nonetheless, these techniques together are shown to reduce the storage requirement by an order of magnitude or more compared to `Cachet`, and to often increase efficiency.

`sharpSAT` also uses a "look ahead" technique known in the SAT community as the *failed literal* rule (the author refers to it as *implicit BCP*). Here every so often one identifies a set of candidate variables for each of which the failed literal test is applied: if setting x to TRUE makes the current formula unsatisfiable, then assert $x =$ FALSE and simplify; otherwise, if setting x to FALSE makes the current formula unsatisfiable, then assert $x =$ TRUE and simplify. The technique is shown to pay off well while model counting several difficult instances.

[5] Note that clause learning and decomposition into components were already implemented in the model counter `Relsat`, but no caching.

Recently, Davies and Bacchus [13] have shown that employing more reasoning at each node of the DPLL search tree can significantly speed-up the model counting process.[6] Specifically, they use hyper-binary resolution and equality reduction in addition to unit propagation, which simplifies the formula and often results in more efficient component detection and caching, and sometimes even stronger component division.

20.2.2. Counting Using Knowledge Compilation

A different approach for exact model counting is to convert or *compile* the given CNF formula into another logical form from which the count can be deduced easily, i.e., in time polynomial in the size of the formula in the new logical form. For example, in principle, one could convert the formula into a binary decision diagram or BDD [8] and then "read off" the solution count by traversing the BDD from the leaf labeled "1" to the root. One advantage of this methodology is that once resources have been spent on compiling the formula into this new form, several complex queries can potentially be answered fairly quickly, often with a linear time traversal with simple book keeping. For instance, with BDDs, one can easily answer queries about satisfiability, being a tautology, logical equivalence to another formula, number of solutions, etc.

A knowledge compilation alternative was introduced by Darwiche [10] in a compiler called `c2d`, which converts the given CNF formula into *deterministic, decomposable negation normal form* or d-DNNF. The DNNF form [9, 12] is a strict superset of ordered BDDs (in the sense that an ordered BDD can be converted in linear time into DNNF), and often more succinct. While a BDD is structurally quite different from a CNF style representation of a Boolean function,[7] the negation normal form or NNF underlying d-DNNF is very much like CNF. Informally, one can think of a CNF formula as a 4-layer directed acyclic graph, with the root node labeled with AND or $\wedge$, pointing to all nodes in the next layer corresponding to clauses and labeled with OR or $\vee$, and each clause node pointing either directly or through a layer-3 node labeled NOT or $\neg$ to nodes labeled with variable names, one for each variable; this represents the conjunction of disjunctions that defines CNF. In contrast, an NNF formula is defined by a rooted directed acyclic graph in which there is no restriction on the depth, each non-leaf node is labeled with either $\wedge$ or $\vee$, each leaf node is labeled with either a variable or its negation, and the leaf nodes only have incoming edges (as before). Thus, unlike CNF, one may have several levels of alternations between $\wedge$ and $\vee$ nodes, but all negations are pushed down all the way to the leaf nodes. There are, in general, twice as many leaf nodes as variables.

In order to exploit NNF for model counting, one must add two features to it, decomposability and determinism:

i. *Decomposability* says that for the children $A_1, A_2, \ldots, A_s$ of a node A^{AND} labeled $\wedge$, the variables appearing in each pair of the A_i's must be disjoint, i.e., the logical expression at an AND-node can be decomposed into disjoint

[6] In SAT solving, this extra reasoning was earlier observed to not be cost effective.

[7] A BDD is more akin to the search space of a DPLL-style process, with nodes corresponding to branching on a variable by fixing it to TRUE or FALSE.

components corresponding to its children. For model counting, this translates into $\#f^{\text{AND}} = \#f_1 \times \#f_2 \times \ldots \times \#f_s$, where f^{AND} and f_i denote the Boolean functions captured by A^{AND} and A_i, respectively.

ii. In a similar manner, *determinism* says that the children $B_1, B_2, \ldots, B_t$ of a node B^{OR} labeled $\vee$ do not have any common solutions, i.e., the logical conjunction of the Boolean functions represented by any two children of an OR-node is inconsistent. For model counting, this translates into $\#f^{\text{OR}} = \#f_1 + \#f_2 + \ldots + \#f_s$, where f^{OR} and f_i denote the Boolean functions captured by B^{OR} and B_i, respectively.

The above properties suggest a simple model counting algorithm which computes $\#F$ from a d-DNNF representation of F, by performing a topological traversal of the underlying acyclic graph starting from the leaf nodes. Specifically, each leaf node is assigned a count of 1, the count of each $\wedge$ node is computed as the product of the counts of its children, and the count of each $\vee$ node is computed as the sum of the counts of its children. The count associated with the root of the graph is reported as the model count of F.

In the simplified d-DNNF form generated by `c2d`, every $\vee$ node has exactly two children, and the node has as its secondary label the identifier of a variable that is guaranteed to appear as TRUE in all solutions captured by one child and as FALSE in all solutions captured by the other child. `c2d` can also be asked to compile the formula into a *smoothed* form, where each child of an $\vee$ node has the same number of variables.

Inside `c2d`, the compilation of the given CNF formula F into d-DNNF is done by first constructing a decomposition tree or *dtree* for F, which is a binary tree whose leaves are tagged with the clauses of F and each of whose non-leaf vertices has a set of variables, called the *separator*, associated with it. The separator is simply the set of variables that are shared by the left and right branches of the node, the motivation being that once these variables have been assigned truth values, the two resulting sub-trees will have disjoint sets of variables. `c2d` uses an exhaustive version of the DPLL procedure to construct the dtree and compile it to d-DNNF, by ensuring that the separator variables for each node are either instantiated to various possible values (and combined using $\vee$ nodes) or no longer shared between the two subtrees (perhaps because of variable instantiations higher up in the dtree or from the resulting unit-propagation simplifications). Once the separator variables are instantiated, the resulting components become disjoint and are therefore combined using $\wedge$ nodes.

`c2d` has been demonstrated to be quite competitive on several classes of formulas, and sometimes more efficient than DPLL-based exact counters like `Cachet` and `Relsat` even for obtaining a single overall model count. For applications that make several counting-related queries on a single formula (such as "marginal probability" computation or identifying "backbone variables" in the solution set), this knowledge compilation approach has a clear "re-use" advantage over traditional DPLL-based counters. This approach is currently being explored also for computing connected clusters in the solution space of a formula.

20.3. Approximate Model Counting

Most exact counting methods, especially those based on DPLL search, essentially attack a #P-complete problem head on—by exhaustively exploring the raw combinatorial search space. Consequently, these algorithms often have difficulty scaling up to larger problem sizes. For example, perhaps it is too much to expect a fast algorithm to be able to precisely distinguish between a formula having 10^{70} and $10^{70} + 1$ solutions. Many applications of model counting may not even care about such relatively tiny distinctions; it may suffice to provide rough "ball park" estimates, as long as the method is quick and the user has some confidence in its correctness. We should point out that problems with a higher solution count are not necessarily harder to determine the model count of. In fact, counters like `Relsat` can compute the true model count of highly under-constrained problems with many "don't care" variables and a lot of models by exploiting big clusters in the solution space. The model counting problem is instead much harder for more intricate combinatorial problems in which the solutions are spread much more finely throughout the combinatorial space.

With an abundance of difficult to count instances, scalability requirements shifted the focus on efficiency, and several techniques for fairly quickly estimating the model count have been proposed. With such estimates, one must consider two aspects: the *quality of the estimate* and the *correctness confidence* associated with the reported estimate. For example, by simply finding one solution of a formula F with a SAT solver, one can easily proclaim with high (100%) confidence that F has at least one solution—a correct lower bound on the model count. However, if F in reality has, say, 10^{15} solutions, this high confidence estimate is of very poor quality. On the other extreme, a technique may report an estimate much closer to the true count of 10^{15}, but may be completely unable to provide any correctness confidence, making one wonder how good the reported estimate actually is. We would ideally like to have some control on both the quality of the reported estimate as well as the correctness confidence associated with it. The quality may come as an empirical support for a technique in terms of it often being fairly close to the true count, while the correctness confidence may be provided in terms of convergence to the true count in the limit or as a probabilistic (or even statistical) guarantee on the reported estimate being a correct lower or upper bound. We have already mentioned in Section 20.1 one such randomized algorithm with strong theoretical guarantees, namely, the FPRAS scheme of Karp and Luby [25]. We discuss in the remainder of this section some approaches that have been implemented and evaluated more extensively.

20.3.1. Estimation Without Guarantees

Using sampling for estimates: Wei and Selman [54] introduced a local search based method that uses Markov Chain Monte Carlo (MCMC) sampling to compute an approximation of the true model count of a given formula. Their model counter, `ApproxCount`, is able to solve several instances quite accurately, while scaling much better than exact model counters as problem size increases.

`ApproxCount` exploits the fact that if one can sample (near-)uniformly from the set of solutions of a formula F, then one can compute a good estimate of

the number of solutions of F.[8] The basic idea goes back to Jerrum, Valiant, and Vazirani [24]. Consider a Boolean formula F with M satisfying assignments. Assuming we could sample these satisfying assignments uniformly at random, we can estimate the fraction of all models that have a variable x set to TRUE, M^+, by taking the ratio of the number of models in the sample that have x set to TRUE over the total sample size. This fraction will converge, with increasing sample size, to the true fraction of models with x set positively, namely, $\gamma = M^+/M$. (For now, assume that $\gamma > 0$.) It follows immediately that $M = (1/\gamma)M^+$. We will call $1/\gamma$ the "multiplier" (> 0). We have thus reduced the problem of counting the models of F to counting the models of a simpler formula, $F^+ = F|_{x=\text{TRUE}}$; the model count of F is simply $1/\gamma$ times the model count of F^+. We can recursively repeat the process, leading to a series of multipliers, until all variables are assigned truth values or—more practically—until the residual formula becomes small enough for us to count its models with an exact model counter. For robustness, one usually sets the selected variable to the truth value that occurs more often in the sample, thereby keeping intact a majority of the solutions in the residual formula and recursively counting them. This also avoids the problem of having $\gamma = 0$ and therefore an infinite multiplier. Note that the more frequently occurring truth value gives a multiplier $1/\gamma$ of at most 2, so that the estimated count grows relatively slowly as variables are assigned values.

In `ApproxCount`, the above strategy is made practical by using an efficient solution sampling method called `SampleSat` [53], which is an extension of the well-known local search SAT solver `Walksat` [46]. Efficiency and accuracy considerations typically suggest that we obtain 20-100 samples per variable setting and after all but 100-300 variables have been set, give the residual formula to an exact model counter like `Relsat` or `Cachet`.

Compared to exact model counters, `ApproxCount` is extremely fast and has been shown to provide very good estimates for solution counts. Unfortunately, there are no guarantees on the uniformity of the samples from `SampleSat`. It uses Markov Chain Monte Carlo (MCMC) methods [31, 33, 28], which often have exponential (and thus impractical) mixing times for intricate combinatorial problems. In fact, the main drawback of Jerrum et al.'s counting strategy is that for it to work well one needs uniform (or near-uniform) sampling, which is a very hard problem in itself. Moreover, biased sampling can lead to arbitrarily bad under- or over-estimates of the true count. Although the counts obtained from `ApproxCount` can be surprisingly close to the true model counts, one also observes cases where the method significantly over-estimates or under-estimates.

Interestingly, the inherent strength of most state-of-the-art SAT solvers comes actually from the ability to quickly narrow down to a certain portion of the search space the solver is designed to handle best. Such solvers therefore sample solutions in a highly non-uniform manner, making them seemingly ill-suited for model counting, unless one forces the solver to explore the full combinatorial space. An intriguing question (which will be addressed in Section 20.3.2) is whether there is

[8] Note that a different approach, in principle, would be to estimate the *density* of solutions in the space of all 2^n truth assignments for an n-variable formula, and extrapolate that to the number of solutions. This would require sampling all truth assignments uniformly and computing how often a sampled assignment is a solution. This is unlikely to be effective in formulas of interest, which have very sparse solution spaces, and is not what `ApproxCount` does.

a way around this apparent limitation of the use of state-of-the-art SAT solvers for model counting.

Using importance sampling: Gogate and Dechter [18] recently proposed a model counting technique called `SampleMinisat`, which is based on sampling from the so-called backtrack-free search space of a Boolean formula through `SampleSearch` [17]. They use an importance re-sampling method at the base. Suppose we wanted to sample all solutions of F uniformly at random. We can think of these solutions sitting at the leaves of a DPLL search tree for F. Suppose this search tree has been made backtrack-free, i.e., all branches that do not lead to any solution have been pruned away. (Of course, this is likely to be impractical to achieve perfectly in practice in reasonably large and complex solution spaces without spending an exponential amount of time constructing the complete tree, but we will attempt to approximate this property.) In this backtrack-free search space, define a random process that starts at the root and at every branch chooses to follow either child with equal probability. This yields a probability distribution on all satisfying assignments of F (which are the leaves of this backtrack-free search tree), assigning a probability of 2^{-d} to a solution if there are d branching choices in the tree from the root to the corresponding leaf. In order to sample not from this particular "backtrack-free" distribution but from the *uniform* distribution over all solutions, one can use the importance sampling technique [42], which works as follows. First sample k solutions from the backtrack-free distribution, then assign a new probability to each sampled solution which is proportional to the inverse of its original probability in the backtrack-free distribution (i.e., proportional to 2^d), and finally sample one solution from this new distribution. As k increases, this process, if it could be made practical, provably converges to sampling solutions uniformly at random.

`SampleMinisat` builds upon this idea, using DPLL-based SAT solvers to construct the backtrack-free search space, either completely or to an approximation. A simple modification of the above uniform sampling method can be used to instead estimate the *number* of solutions (i.e., the number of leaves in the backtrack-free search space) of F. The process is embedded inside a DPLL-based solver, which keeps track of which branches of the search tree have already been shown to be unsatisfiable. As more of the search tree is explored to generate samples, more branches are identified as unsatisfiable, and one gets closer to achieving the exact backtrack-free distribution. In the limit, as the number of solution samples increases to infinity, the entire search tree is explored and all unsatisfiable branches tagged, yielding the true backtrack-free search space. Thus, this process in the limit converges to purely uniform solutions and an accurate estimate of the number of solutions. Experiments with `SampleMinisat` show that it can provide very good estimates of the solution count when the formula is within the reach of DPLL-based methods. In contrast, `ApproxCount` works well when the formula is more suitable for local search techniques like `Walksat`.

Gogate and Dechter [18] show how this process, when using the exact backtrack-free search space (as opposed to its approximation), can also be used to provide lower bounds on the model count with probabilistic correctness guarantees following the framework of `SampleCount`, which we discuss next.

20.3.2. Lower and Upper Bounds With Guarantees

Using sampling for estimates with guarantees: Building upon `ApproxCount`, Gomes et al. [19] showed that, somewhat surprisingly, using sampling with a modified, randomized strategy, one can get provable *lower bounds* on the total model count, with high confidence (probabilistic) correctness guarantees, *without any requirement on the quality of the sampling process*. They provide a formal analysis of the approach, implement it as the model counter `SampleCount`, and experimentally demonstrate that it provides very good lower bounds—with high confidence and within minutes—on the model counts of many problems which are completely out of reach of the best exact counting methods. The key feature of `SampleCount` is that the correctness of the bound reported by it holds even when the sampling method used is arbitrarily bad; only the quality of the bound may deteriorate (i.e., the bound may get farther away from the true count on the lower side). Thus, the strategy remains sound even when a heuristic-based practical solution sampling method is used instead of a true uniform sampler.

The idea is the following. Instead of using the solution sampler to select the variable setting and to compute a multiplier, let us use the sampler only as a heuristic to determine *in what order* to set the variables. In particular, we will use the sampler to select a variable whose positive and negative setting occurs most balanced in our set of samples (ties are broken randomly). Note that such a variable will have the highest possible multiplier (closest to 2) in the `ApproxCount` setting discussed above. Informally, setting the most balanced variable will divide the solution space most evenly (compared to setting one of the other variables). Of course, our sampler may be heavily biased and we therefore cannot really rely on the observed ratio between positive and negative settings of a variable. Interestingly, we can simply set the variable to a randomly selected truth value and use the multiplier 2. This strategy will still give—in expectation—the true model count. A simple example shows why this is so. Consider the formula F used in the discussion in Section 20.3.1 and assume x occurs most balanced in our sample. Let the model count of F^+ be $2M/3$ and of F^- be $M/3$. If we decide with probability $1/2$ to set x to TRUE, we obtain a total model count of $2 \times 2M/3$, i.e., too high; but, with probability $1/2$, we will set x to FALSE, obtaining a total count of $2 \times M/3$, i.e., too low. Together, these imply an expected (average) count of exactly M.

Technically, the *expected total model count* is correct because of the linearity of expectation. However, we also see that we may have significant *variance* between specific counts, especially since we are setting a series of variables in a row (obtaining a sequence of multipliers of 2), until we have a simplified formula that can be counted exactly. In fact, in practice, the distribution of the estimated total count (over different runs) is often heavy-tailed [27]. To mitigate the fluctuations between runs, we use our samples to select the "best" variables to set next. Clearly, a good heuristic would be to set such "balanced" variables first. We use `SampleSat` to get guidance on finding such balanced variables. The random value setting of the selected variable leads to an expected model count that is equal to the actual model count of the formula. Gomes et al. [19] show how this property can be exploited using Markov's inequality to obtain lower bounds on

the total model count with predefined confidence guarantees. These guarantees can be made arbitrarily strong by repeated iterations of the process.

What if all variable are found to be not so balanced? E.g., suppose the most balanced variable x is TRUE in 70% of the sampled solutions and FALSE in the remaining 30%. While one could still set x to TRUE or FALSE with probability 1/2 each as discussed above and still maintain a correct count in expectation, Kroc et al. [29] discuss how one might reduce the resulting variance by instead using a *biased coin* with probability $p = 0.7$ of Heads, setting x to TRUE with probability 0.7, and scaling up the resulting count by $1/0.7$ if x was set to TRUE and by $1/0.3$ if x was set to FALSE. If the samples are uniform, this process provably reduces the variance, compared to using an unbiased coin with $p = 0.5$.

The effectiveness of `SampleCount` is further boosted by using variable "equivalence" when no single variable appears sufficiently balanced in the sampled solutions. For instance, if variables x_1 and x_2 occur with the same polarity (either both TRUE or both FALSE) in nearly half the sampled solutions and with a different polarity in the remaining, we randomly replace x_2 with either x_1 or $\neg x_1$, and simplify. This turns out to have the same simplification effect as setting a single variable, but is more advantageous when no single variable is well balanced.

Using XOR*-streamlining:* `MBound` [21] is a very different method for model counting, which interestingly uses any complete SAT solver "as is" in order to compute an estimate of the model count of the given formula. It follows immediately that the more efficient the SAT solver used, the more powerful this counting strategy becomes. `MBound` is inspired by recent work on so-called "streamlining constraints" [22], in which additional, non-redundant constraints are added to the original problem to increase constraint propagation and to focus the search on a small part of the subspace, (hopefully) still containing solutions. This technique was earlier shown to be successful in solving very hard combinatorial design problems, with carefully created, domain-specific streamlining constraints. In contrast, `MBound` uses a domain-independent streamlining process, where the streamlining constraints are constructed purely at random.

The central idea of the approach is to use a special type of randomly chosen constrains, namely XOR or parity constraints on the original variables of the problem. Such constraints require that an *odd* number of the involved variables be set to TRUE. (This requirement can be translated into the usual CNF form by using additional variables [50], and can also be modified into requiring that an *even* number of variables be TRUE by adding the constant 1 to the set of involved variables.)

`MBound` works in a very simple fashion: repeatedly add a number s of purely random XOR constraints to the formula as additional CNF clauses, feed the resulting streamlined formula to a state-of-the-art complete SAT solver without any modification, and record whether or not the streamlined formula is still satisfiable. At a very high level, each random XOR constraint will cut the solution space of satisfying assignments approximately in half. As a result, intuitively speaking, if after the addition of s XOR constraints the formula is still satisfiable, the original formula must have at least of the order of 2^s models. More rigorously, it can be shown that if we perform t experiments of adding s random XOR constraints

and our formula remains satisfiable in each case, then with probability at least $1 - 2^{-\alpha t}$, our original formula will have at least $2^{s-\alpha}$ satisfying assignments for any $\alpha > 0$, thereby obtaining a lower bound on the model count with a probabilistic correctness guarantee. The confidence expression $1 - 2^{-\alpha t}$ says that by repeatedly doing more experiments (by increasing t) or by weakening the claimed bound of $2^{s-\alpha}$ (by increasing α), one can arbitrarily boost the confidence in the lower bound count reported by this method.

The method generalizes to the case where some streamlined formulas are found to be satisfiable and some are not. Similar results can also be derived for obtaining an upper bound on the model count, although the variance-based analysis in this case requires that the added XOR constraints involve as many as $n/2$ variables on average, which often decreases the efficiency of SAT solvers on the streamlined formula.[9] The efficiency of the method can be further boosted by employing a *hybrid* strategy: instead of feeding the streamlined formula to a SAT solver, feed it to an exact model counter. If the exact model counter finds M solutions to the streamlined formula, the estimate of the total model count of the original formula is taken to be $2^{s-\alpha} \times M$, again with a formal correctness probability attached to statistics over this estimate over several iterations; the minimum, maximum, and average values over several iterations result in different associated correctness confidence.

A surprising feature of this approach is that it does not depend at all on how the solutions are distributed throughout the search space. It relies on the very special properties of random parity constraints, which in effect provide a good hash function, randomly dividing the solutions into two near-equal sets. Such constraints were earlier used by Valiant and Vazirani [52] in a randomized reduction from SAT to the related problem UniqueSAT (a "promise problem" where the input is guaranteed to have either 0 or 1 solution, never 2 or more), showing that UniqueSAT is essentially as hard as SAT. Stockmeyer [47] had also used similar ideas under a theoretical setting. The key to converting this approach into a state-of-the-art model counter was the relatively recent observation that very short XOR constraints—the only ones that are practically feasible with modern constraint solvers—can provide a fairly accurate estimate of the model count on a variety of domains of interest [21, 20].

Exploiting Belief Propagation methods: In recent work, Kroc et al. [29] showed how one can use probabilistic reasoning methods for model counting. Their algorithm, called `BPCount`, builds upon the lower bounding framework of `SampleCount`. However, instead of using several solution samples for heuristic guidance—which can often be time consuming—they use a probabilistic reasoning approach called Belief Propagation or BP. A description of BP methods in any reasonable detail is beyond the scope of this chapter; we refer the reader to standard texts on this subject [e.g. 39] as well as to Part 2, Chapter 18 of this Handbook. In essence, BP is a general "message passing" procedure for

[9] It was later demonstrated empirically [20] that for several problem domains, significantly shorter XOR constraints are effectively as good as XOR constraints of length $n/2$ ("full length") in terms of the key property needed for the accuracy of this method: low *variance* in the solution count estimate over several runs of `MBound`.

probabilistic reasoning, and is often described in terms of a set of mutually recursive equations which are solved in an iterative manner. On SAT instances, BP works by passing likelihood information between variables and clauses in an iterative fashion until a fixed point is reached. From this fixed point, statistical information about the solution space of the formula can be easily computed. This statistical information is provably exact when the constraint graph underlying the formula has a tree-like structure, and is often reasonably accurate (empirically) even when the constraint graph has cycles [cf. 35].

For our purposes, BP, in principle, provides *precisely* the information deduced from solution samples in `SampleCount`, namely, an estimate of the *marginal probability* of each variable being TRUE or FALSE when all solutions are sampled uniformly at random. Thus, `BPCount` works exactly like `SampleCount` and provides the same probabilistic correctness guarantees, but is often orders of magnitude faster on several problem domains because running BP on the formula is often much faster than obtaining several solution samples through `SampleSat`. A challenge in this approach is that the standard mutually recursive BP equations don't even converge to any fixed point on many practical formulas of interest. To address this issue, Kroc et al. [29] employ a *message-damping* variant of BP whose convergence behavior can be controlled by a continuously varying parameter. They also use *safety checks* in order to avoid fatal mistakes when setting variables (i.e., to avoid inadvertently making the formula unsatisfiable).

We note that the model count of a formula can, in fact, also be estimated directly from just one fixed point run of the BP equations, by computing the value of so-called partition function [55]. However, the count estimated this way is often highly inaccurate on structured loopy formulas. `BPCount`, on the other hand, makes a much more robust use of the information provided by BP.

Statistical upper bounds: In a different direction, Kroc et al. [29] propose a second method, called `MiniCount`, for providing *upper bounds* on the model counts of formulas. This method exploits the power of modern DPLL-based SAT solvers, which are extremely good at finding single solutions to Boolean formulas through backtrack search. The problem of computing upper bounds on the model count had so far eluded solution because of the asymmetry (discussed earlier in the partial counts sub-section of Section 20.2.1) which manifests itself in at least two inter-related forms: the set of solutions of interesting n variable formulas typically forms a minuscule fraction of the full space of 2^n truth assignments, and the application of Markov's inequality as in the lower bound analysis of `SampleCount` does not yield interesting upper bounds. As noted earlier, this asymmetry also makes upper bounds provided by partial runs of exact model counters often not very useful. To address this issue, Kroc et al. [29] develop a *statistical framework* which lets one compute upper bounds under certain statistical assumptions, which are independently validated.

Specifically, they describe how the SAT solver `MiniSat` [14], with two minor modifications—randomizing whether the chosen variable is set to TRUE or to FALSE, and disabling restarts—can be used to estimate the total number of solutions. The number d of branching decisions (not counting unit propagations and failed branches) made by `MiniSat` before reaching a solution, is the main

quantity of interest: when the choice between setting a variable to TRUE or to FALSE is randomized,[10] the number d is provably not any lower, in expectation, than $\log_2$ of the true model count. This provides a strategy for obtaining upper bounds on the model count, only if one could efficiently estimate the expected value, $\mathbb{E}[d]$, of the number of such branching decisions. A natural way to estimate $\mathbb{E}[d]$ is to perform multiple runs of the randomized solver, and compute the average of d over these runs. However, if the formula has many "easy" solutions (found with a low value of d) and many "hard" solutions (requiring large d), the limited number of runs one can perform in a reasonable amount of time may be insufficient to hit many of the "hard" solutions, yielding too low of an estimate for $\mathbb{E}[d]$ and thus an incorrect upper bound on the model count.

Interestingly, they show that for many families of formulas, d has a distribution that is very close to the normal distribution; in other words, the estimate 2^d of the upper bound on the number of solutions is log-normally distributed. Now, under the assumption that d is indeed normally distributed, estimating $\mathbb{E}[d]$ for an upper bound on the model count becomes easier: when sampling various values of d through multiple runs of the solver, one need not necessarily encounter both low and high values of d in order to correctly estimate $\mathbb{E}[d]$. Instead, even with only below-average samples of d, the assumption of normality lets one rely on standard statistical tests and conservative computations to obtain a statistical upper bound on $\mathbb{E}[d]$ within any specified confidence interval. We refer the reader to the full paper for details. Experimentally, `MiniCount` is shown to provide good upper bounds on the solution counts of formulas from many domains, often within seconds and fairly close to the true counts (if known) or separately computed lower bounds.

20.4. Conclusion

With SAT solving establishing its mark as one of the most successful automated reasoning technologies, the model counting problem for SAT has begun to see a surge of activity in the last few years. One thing that has been clear from the outset is that model counting is a much harder problem. Nonetheless, thanks to its broader scope and applicability than SAT solving, it has led to a range of new ideas and approaches—from DPLL-based methods to local search sampling estimates to knowledge compilation to novel randomized streamlining methods. Practitioners working on model counting have discovered that many interesting techniques that were too costly for SAT solving are not only cost effective for model counting but crucial for scaling practical model counters to reasonably large instances. The variety of tools for model counting is already rich, and growing. While we have made significant progress, at least two key challenges remain open: how do we push the limits of scalability of model counters even further, and how do we extend techniques to do model counting for *weighted* satisfiability problems?[11] While exact model counters are often able to also solve

[10] `MiniSat` by default always sets variables to FALSE.

[11] In weighted model counting, each variable x has a weight $p(x) \in [0, 1]$ when set to TRUE and a weight $1 - p(x)$ when set to FALSE. The weight of a truth assignment is the product of the weights of its literals. The weighted model count of a formula is the sum of the weights of

the weighted version of the problem at no extra cost, much work needs to be done to adapt the more scalable approximate methods to handle weighted model counting.

References

[1] D. Angluin. On counting problems and the polynomial-time hierarchy. *Theoretical Computer Science*, 12:161–173, 1980.

[2] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings of FOCS-03: 44th Annual Symposium on Foundations of Computer Science*, pages 340–351, Cambridge, MA, Oct. 2003.

[3] D. D. Bailey, V. Dalmau, and P. G. Kolaitis. Phase transitions of PP-complete satisfiability problems. *Discrete Applied Mathematics*, 155(12): 1627–1639, 2007.

[4] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, Austin, TX, July 2000.

[5] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97: 14th National Conference on Artificial Intelligence*, pages 203–208, Providence, RI, July 1997.

[6] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proceedings 18th Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.

[7] E. Birnbaum and E. L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.

[8] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[9] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48 (4):608–647, 2001.

[10] A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of ECAI-04: 16th European Conference on Artificial Intelligence*, pages 328–332, Valencia, Spain, Aug. 2004.

[11] A. Darwiche. The quest for efficient probabilistic inference, July 2005. Invited Talk, IJCAI-05.

[12] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[13] J. Davies and F. Bacchus. Using more reasoning to improve #SAT solving. In *Proceedings of AAAI-07: 22nd National Conference on Artificial Intelligence*, pages 185–190, Vancouver, BC, July 2007.

[14] N. Eén and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, St. Andrews, U.K., June 2005.

its satisfying assignments.

[15] E. Fischer, J. A. Makowsky, and E. V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] V. Gogate and R. Dechter. A new algorithm for sampling CSP solutions uniformly at random. In *CP-06: 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 711–715, Nantes, France, Sept. 2006.

[18] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proceedings of AAAI-07: 22nd National Conference on Artificial Intelligence*, pages 198–203, Vancouver, BC, July 2007.

[19] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *Proceedings of IJCAI-07: 20th International Joint Conference on Artificial Intelligence*, pages 2293–2299, Hyderabad, India, Jan. 2007.

[20] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. Short XORs for model counting; from theory to practice. In *Proceedings of SAT-07: 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 100–106, Lisbon, Portugal, May 2007.

[21] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, pages 54–61, Boston, MA, July 2006.

[22] C. P. Gomes and M. Sellmann. Streamlined constraint reasoning. In *CP-04: 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 274–289, Toronto, Canada, Oct. 2004.

[23] G. Gottlob, F. Scarcello, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.

[24] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[25] R. M. Karp and M. Luby. Monte-Carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, 1(1):45–64, 1985.

[26] R. M. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.

[27] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of AAAI-06: 21st National Conference on Artificial Intelligence*, pages 1014–1019, Boston, MA, July 2006.

[28] S. Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by simuleated annealing. *Science*, 220(4598):671–680, 1983.

[29] L. Kroc, A. Sabharwal, and B. Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *CPAIOR-08: 5th International Conference on Integration of AI and OR Techniques in Constraint Programming*, volume 5015 of *Lecture Notes in Computer Science*, pages

127–141, Paris, France, May 2008.

[30] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.

[31] N. Madras. Lectures on Monte Carlo methods. In *Field Institute Monographs*, volume 16. American Mathematical Society, 2002.

[32] S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of AAAI-98: 15th National Conference on Artificial Intelligence*, pages 954–959, Madison, WI, July 1998.

[33] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[34] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC-01: 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[35] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of UAI-99: 15th Conference on Uncertainty in Artificial Intelligence*, pages 467–475, Sweden, July 1999.

[36] N. Nishimura, P. Ragde, and S. Szeider. Solving #SAT using vertex covers. In *Proceedings of SAT-06: 9th International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 396–409, Seattle, WA, Aug. 2006.

[37] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[38] J. D. Park. MAP complexity results and approximation methods. In *Proceedings of UAI-02: 18th Conference on Uncertainty in Artificial Intelligence*, pages 388–396, Edmonton, Canada, Aug. 2002.

[39] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[40] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of SAT-07: 10th International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299, Lisbon, Portugal, May 2007.

[41] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[42] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley & Sons, 1981.

[43] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of SAT-04: 7th International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, May 2004.

[44] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240, St. Andrews, U.K., June 2005.

[45] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proceedings of AAAI-05: 20th National Con-*

ference on Artificial Intelligence, pages 475–482, Pittsburgh, PA, July 2005.

[46] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.

[47] L. J. Stockmeyer. On approximation algorithms for #P. *SIAM Journal on Computing*, 14(4):849–861, 1985.

[48] M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proceedings of SAT-06: 9th International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429, Seattle, WA, Aug. 2006.

[49] S. Toda. On the computational power of PP and $\oplus$P. In *FOCS-89: 30th Annual Symposium on Foundations of Computer Science*, pages 514–519, 1989.

[50] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*. 1968.

[51] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[52] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(3):85–93, 1986.

[53] W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of AAAI-04: 19th National Conference on Artificial Intelligence*, pages 670–676, San Jose, CA, July 2004.

[54] W. Wei and B. Selman. A new approach to model counting. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339, St. Andrews, U.K., June 2005.

[55] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on Information Theory*, 51(7):2282–2312, 2005.

Index

#P, 2

backtrack-free search space, 13
belief propagation, 16
binary decision diagram, 9
BP, *see* belief propagation

caching, 7
compilation, *see* knowledge compilation
component caching, *see* caching
components, 6

d-DNNF, 9
decomposable negation normal form, 9
DPLL, 5

equality reduction, 9

failed literal, 8
formula caching, *see* caching
FPRAS, 4

hyper-binary resolution, 9

implicit BCP, 8
importance sampling, 13

knowledge compilation, 9

look ahead, 8

Markov Chain Monte Carlo (MCMC), 4, 11
model counter
  ApproxCount, 12
  BPCount, 17
  c2d, 9
  Cachet, 8
  CDP, 5
  MBound, 15
  MiniCount, 17
  Relsat, 6
  SampleCount, 14
  SampleMinisat, 13
  sharpSAT, 8
  XOR streamlining, 15
model counting, 1
  approximate, 11
  exact, 4
  FPRAS, 4

number P, *see* #P

parity constraint, 15
partition function, 17
permanent, 3
phase transition, 7
PP, 4
PSPACE, 4

quantified Boolean formula, 3

random problems
  3-SAT, 7
reduction
  counting, 3
  parsimonious, 3

sharp P, *see* #P
solution counting, *see* model counting
solution sampling, 11, 13

XOR constraint, 15