A New Approach to Model Counting

Wei Wei and Bart Selman

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract. We introduce ApproxCount, an algorithm that approximates the number of satisfying assignments or models of a formula in propositional logic. Many AI tasks, such as calculating degree of belief and reasoning in Bayesian networks, are computationally equivalent to model counting. It has been shown that model counting in even the most restrictive logics, such as Horn logic, monotone CNF and 2CNF, is intractable in the worst-case. Moreover, even approximate model counting remains a worst-case intractable problem. So far, most practical model counting algorithms are based on backtrack style algorithms such as the DPLL procedure. These algorithms typically yield exact counts but are limited to relatively small formulas. Our ApproxCount algorithm is based on SampleSat, a new algorithm that samples from the solution space of a propositional logic formula near-uniformly. We provide experimental results for formulas from a variety of domains. The algorithm produces good estimates for formulas much larger than those that can be handled by existing algorithms.

1 Introduction

In recent years, we have seen tremendous improvements in our abilities to solve large Satisfiability (SAT) problems. The field benefits from significant progress both in terms of our theoretical understanding of the SAT problem, and in terms of practical algorithm design and engineering of solvers. Because the state-of-the-art SAT solvers can often handle formulas with thousands of variables [19], problems from many other domains, such as planning [8] and microprocessor verification [16], have been encoded to SAT instances, and solved effectively by SAT solvers. This indirect approach is often competitive with, if not faster than, solving the problems directly with methods developed specifically for the domain under consideration.

Broadly speaking the “SAT approach” appears to work quite well for NP-complete problems. However, many tasks in AI belong to higher complexity classes than NP, and therefore, presumably, cannot be encoded as SAT problems, without introducing exponentially many variables. A number of these tasks are computationally equivalent to counting the satisfying assignments of a formula in propositional logic [11]. For example, consider a knowledge base (KB) written in propositional logic with no explicit probabilistic information and a statement s. The degree of belief in s, which is defined as $P(s|KB)$, can be calculated by $\frac{\mathcal{M}(s \wedge KB)}{\mathcal{M}(KB)}$, where $\mathcal{M}(\cdot)$ is the model (satisfying assignment) count of the input formula. Moreover, probabilistic reasoning using Bayesian belief networks can also to be reduced effectively to counting models of Boolean formulas.

It has been shown that many techniques developed for SAT, such as DPLL [4] and clause learning [9], can be adapted to solve model counting problems. However the memory usage and run time of such algorithms often increase exponentially with problem size, and such algorithms are therefore limited to relatively small formulas.

In this paper, we present ApproxCount, an approximate model counting algorithm based on a biased random walk strategy. Random walk strategies have been shown to be effective on a variety of SAT instances [13]. Recently, a biased random walk strategy was used in SampleSat [17], a new algorithm that samples from the solution space of a Boolean formula near-uniformly. Following the scheme outlined by Jerrum *et al* [7], ApproxCount uses SampleSat to repetitively draw samples from the solution spaces of a Boolean formula and its sub-formulas.

A key advantage of a sampling approach for counting over approaches based on complete backtrack search is that the run time can be controlled much better. After the first several samples (satisfying assignments) are drawn, we can obtain a reasonably accurate estimate of total run time depending on how accurate we want our final count to be. In applications where a decision has to be made by a certain deadline, the algorithm can adapt by drawing fewer samples in each iteration.

The number of calls ApproxCount makes to SampleSat is determined by the number of variables in the formula under consideration. An intriguing question is whether the error in each intermediate result due to SampleSat's deviation from uniformly sampling will accumulate to an unmanageably large overall error rate. After all, even a small error in each step can potentially lead to an exponential deviation from the exact count, which would make our approach of little practical use. Somewhat surprisingly, our experiments show that the errors from setting different variables appear to offset each other, resulting in an overall final estimate that is within a factor two of the exact count in many domains, including random 3CNF formulas, several structured problems, and constructed combinatorial instances.

For random 3CNF formulas, the complexity of model counting is determined by the clause/variable (C/V) ratio. Low C/V ratios are particularly difficult for counting procedures because of the very large number of satisfying assignments. In recent years, in work on DPLL based model counters, the C/V ratio that requires the largest run time has shifted from 1.2 to 2.0, with the introduction of new techniques and algorithms. In our work, we show that the error rate of ApproxCount on random 3CNF formulas stays relatively constant for different C/V ratios, when given the same total run time. This suggests that our approach is more robust compared to the DPLL style techniques.

The difficulty in evaluating our ApproxCount algorithm is that exact model counting programs often do not work on larger instances. We therefore introduce a class of structured formulas, where we know the exact number of satisfying assignments (obtained from basic combinatorics). For DPLL style methods, the run time explodes exponentially with the problem size. However, we will show that ApproxCount approximates the model counts will high accuracy, while the run time scales only polynomially with problem size.

The reminder of this paper is organized as follows. Background with theoretical results and development of existing model counting approaches is discussed in the next section. In Section 3, we review the SampleSat algorithm. Section 4 provides a detailed description of our new approximate model counting algorithm, and discusses on several implementation issues. Experimental results are given in Section 5. We then discuss how to extend our algorithm to deal with real numbers in probabilistic reasoning. In the last section, we summarize the main results of this paper.

2 Background

Before we start our discussion, we first define some terms that we use throughout this paper. Without loss of generality, the Boolean formulas we discuss are in their conjunction normal form (CNF) unless otherwise indicated. A CNF is a conjunction of clauses. A clause is defined as a disjunction of literals. A literal is a Boolean variable, which ranges over $\{True, False\}$, or its negation. When an assignment of truth values to the Boolean variables makes a formula evaluate to *True*, the formula is said to be satisfied, and the assignment is called a satisfying assignment, or interchangeably, a model of the formula. $\mathcal{M}(F)$ denotes the number of unique satisfying assignments of formula F.

2.1 Complexity Results

The problem of counting the satisfying assignments of a propositional logic formula is #P-complete [15], a complexity class that is at least as hard as the polynomial-time hierarchy [14]. To approximate the model count within δ, we look for a number M that satisfies

$$M/(1+\delta) \leq \mathcal{M}(F) \leq M(1+\delta).$$

For any positive δ, approximating the model count of an arbitrary Boolean formula F is NP-hard. This is straightforward because given an approximate counting oracle, one can determine the satisfiablity of F by the positivity of M. $M > 0$ is equivalent to F being satisfiable.

However, it is somewhat surprising that counting and approximate counting of some most restrictive classes of propositional logic are intractable in the worst case. One of such examples is 2MONCNF, a subset of both monotone CNF (all variables occur positively) and 2CNF. Determining satisfiability of 2MONCNF is trivial because each formula in 2MONCNF is satisfied by assigning truth value *True* to each variable. Counting the models of 2MONCNF is shown to be #P-complete, and it is NP-hard to approximate the count to a factor of $2^{n^{1-\epsilon}}$ for any positive constant ϵ [11]. Note approximating to a factor of 2^n is equivalent to satisfiability problem.

Because of these strong negative results, for any interesting classes of propositional logic, guaranteed approximate counting algorithms that run in worst-case polynomial time are not expected to exist. There must be tradeoffs between the approximate factor guarantee and the worst-case run time guarantee. In practice, most existing algorithms sacrifice the run time guarantee to achieve a guaranteed approximation, which is often the exact count.

2.2 Counting by DPLL Searching

DPLL algorithm [4] was designed to find a satisfying assignment of a Boolean formula. The basic strategy of DPLL is that instead of looking for a satisfying assignment of the original formula F directly, one chooses any variable α that appears in F, and a satisfying assignment of either formula $(F \wedge \alpha)$ or formula $(F \wedge \neg\alpha)$ is also a satisfying assignment of F. The two sub-formulas can be simplified by unit-propagation, and solved recursively by calling DPLL algorithm.

The algorithm can be extended to a model counting algorithm in the following way [2]. Because the satisfying assignments of formula $(F \wedge \alpha)$ and formula $(F \wedge \neg\alpha)$ are disjoint,

$$\mathcal{M}(F) = \mathcal{M}(F \wedge \alpha) + \mathcal{M}(F \wedge \neg\alpha).$$

It is observed that when the two sub-formulas are simplified by unit-propagation, variables that appear in F may not appear in the simplified version of $(F \wedge \alpha)$ or $(F \wedge \neg\alpha)$. To expedite the search, if there are n variables in formula F, n^+ variables in Unitprop$(F \wedge \alpha)$, and n^- variables in Unitprop$(F \wedge \neg\alpha)$, then

$$\mathcal{M}(F) = \mathcal{M}(\text{Unitprop}(F \wedge \alpha)) \cdot 2^{n-n^+} + \mathcal{M}(\text{Unitprop}(F \wedge \neg\alpha)) \cdot 2^{n-n^-}.$$

In 2000, the idea of connected component analysis was introduced by Bayardo and Pehoushek [1] as an enhancement to Relsat's model counting ability. They draw connectivity graph for the formula. Nodes in the graph are variables in the formula. Two nodes are connected if the corresponding variables appear in the same clause. If a formula F can be decomposed to sub-formulas $F_1, F_2, \ldots, F_i$, which are induced by the connected components of its connectivity graph, then

$$\mathcal{M}(F) = \prod_{j=1}^{i} \mathcal{M}(F_j).$$

Therefore before picking a variable and branching on it as described previously, a formula is divided into components, and the models of each component is counted by a DPLL model counter recursively.

The most recent additions to DPLL model counting are the ideas of component caching and clause learning [12]. Component caching records the previous counted sub-problems, and therefore avoids counting the same components repetitively. Clause learning is much like that of ZChaff [9], where reasons of previously discovered conflicts are captured in new clauses, and the learned clauses are added to the original formula to avoid running into the same conflicts again. The introduction of these two powerful tools accelerate the model counting procedure by orders of magnitude.

2.3 Counting by Compiling

Another existing approach to model counting is to compile CNF formulas to logics for which counting operation is tractable. Examples of such logics include Ordered Binary Decision Diagrams (OBDD) [6], and its superset, Deterministic, Decomposable Negation Normal Form (d-DNNF) [3]. Model counting is a polynomial operation for both OBDD and d-DNNF, but CNFs cannot always be compiled to these forms of polynomial size. Although this approach is conceptually different from that of DPLL, the actual computation is often similar. For example, the search tree constructed by DPLL with component analysis can be view as a tree-structured d-DNNF, and the component caching idea corresponds to reuse of NNF fragments in non-tree-structured d-DNNF. Huang and Darwiche [6] show that DPLL algorithm can be used to compile CNFs to OBDDs efficiently, and techniques in SAT solvers, such as clause learning and unit propagation, can be utilized in the compiling process.

2.4 Our Proposal: Counting by Sampling

Since model counting problem is downward self-reducible [7], meaning that it can be solved using oracles to solve its sub-problems, approximate counting of solutions can be reduced to almost-uniform sampling of the solution space in polynomial time. ApproxCount algorithm is based on near-uniform sampling. One advantage of using sampling-based approximate counter is that the run time and accuracy are based on the number of samples the algorithm draws in each iteration. One can get a faster (and less accurate) approximation by reducing sample size. This is especially important in time-critical decision making. In searching and compiling based model counters, one cannot halt the execution and retrieve an approximation since it is very possible that many solutions reside in the final branches of the search tree, and, in case of connected component analysis, non-existent of solutions in the last component implies non-existent of solutions in the whole formula.

The best known methods for sampling from a predefined distribution in combinatorial space are Markov Chain Monte Carlo (MCMC) methods. MCMC methods set the target distribution as the stationary distribution of an ergodic Markov chain. With infinite time, the constructed Markov chain is guaranteed to reach its stationary distribution. However, in practice for complex combinatorial problems such as SAT, the Markov chain almost always takes exponential time to reach its stationary distribution [17]. The most widely used sampling algorithms such as Gibbs sampling are often trapped in modes (local minima) and does not converge in practical time limit.

For this reason, we built ApproxCount on SampleSat algorithm [17]. SampleSat is capable of sampling from solution space of a propositional logic near-uniformly and efficiently. We will first briefly review SampleSat algorithm, and then we will describe our implementation of an approximate counter based on it.

3 SampleSat

SampleSat algorithm is based on random walk strategies widely used in solving satisfiability problem [13, 18]. The inherent randomness in random walk style SAT solvers often leads the algorithms to different models in different runs. This provides a biased sampling of solution space. To reduce this bias, MCMC moves, more specifically, Metropolis moves are injected to interleave with random walk moves. This hybrid approach makes the sampling much more uniform, and is useful in many domains, including approximate model counting discussed in this paper.

3.1 Random Walk

Random walk (RW) as a local search heuristic was first proposed by Papadimitriou [10]. The algorithm starts from a random truth assignment. If the assignment has not already satisfied the formula, at each step, one unsatisfied clause is chosen uniformly at random. And then a variable in the clause is chosen by some heuristic **ChooseVar**. The value of the variable is flipped. The algorithm repeats these steps until a satisfying assignment is reached.

```
Procedure RW
repeat
    $c$:= an unsatisfied clause chosen at random
    $x$:= a variable in $c$ chosen by heuristic ChooseVar($c$)
    flip the value of $x$;
until a satisfying assignment is found.
```

Fig. 1. Random walk strategies.

It has been shown that when **ChooseVar**(c) is "picking a variable in c uniformly at random", the RW procedure solves any 2CNF instance in quadratic time with high probability. For more general instances however, the algorithm needs to adopt a heuristic with greedy bias, which tries to satisfy more clauses on each flip. Therefore, the concept of "break value" of a variable is introduced. The break value of a variable is defined by the number of clauses that are currently satisfied but become unsatisfied when the truth value of the said variable is changed. In SampleSat, we follow the heuristic used in WalkSat [13], and define **ChooseVar** as in Figure 2.

```
Heuristic ChooseVar($c$)
if there exists a variable $x$ in $c$ with break value = 0
    return variable $x$
else
    with probability q
        $x$:= a variable in $c$ chosen at random;
        return variable $x$
    with probability (1-q)
        $x$:= a variable in $c$ with smallest break value
        return variable $x$
```

Fig. 2. Heuristic ChooseVar used in WalkSat and SampleSat.

With multiple runs, we found that WalkSat is able to reach every solution of instances from many domains, such as random 3CNF, planning, and verification. However, the sampling is biased, especially for random 3CNF formulas. For example, we found in our experiments on a 70-variable random 3CNF formula near the transition point that the most frequently visited solution is reached 17,000 times more often than the least frequently visited solution. From the theoretically point of view, it is possible to construct a formula, for which a random walk strategy visits one model exponentially more often than it visits another model [17]. Intuitively, a model whose neighbors are all models of the formula cannot be reached by random walk unless it was hit by the initial guess of the algorithm. To reduce the bias and make the sampling more uniform, we incorporate Metropolis moves to the algorithm.

3.2 Metropolis

Metropolis moves are injected into random walk moves because of their favorable limiting properties. The limiting distribution of Metropolis algorithm is uniformly distributed over all models. Metropolis algorithm determines the change of a variable assignment by the decrease in the number of satisfied clauses, Δcost, caused by the change and a predetermined constant T representing temperature. The detail of our Metropolis move implementation is given in Figure 3.

```
Metropolis Algorithm
$x$:= a variable chosen uniformly at random;
if $\Delta\text{cost}(x) \leq 0$
    flip the value of $x$;
else
    with probability $e^{-\Delta\textbf{cost}/T}$
        flip the value of $x$.
```

Fig. 3. Metropolis moves.

SampleSat combines random walk moves with Metropolis moves. At each step, the algorithm makes a random walk move with probability p, and it makes a Metropolis move with probability $(1-p)$. Experiments show $p = 50\%$ yields good sampling results in many domains.

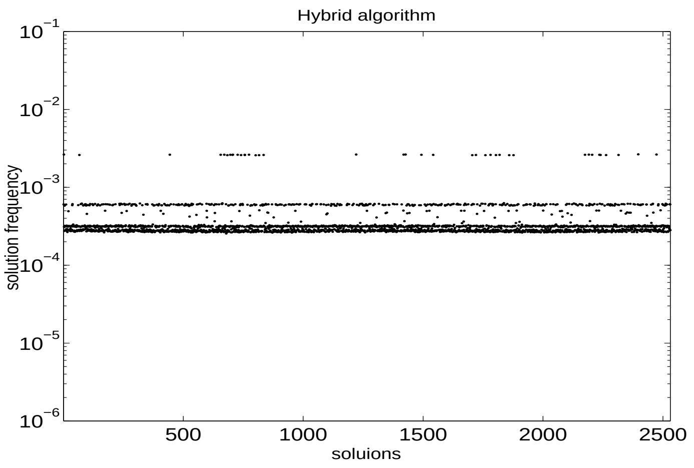

Fig. 4. Sampling of a hard 70-variable random 3SAT instance using SampleSat. Source: [17]

Figure 4 shows the sampling result of this hybrid algorithm on the 70-variable random formula mentioned in Section 3.1. The instance has 2531 solutions. Each dot in the figure presents one solution. The y-axis represents the solution frequency (#hits/#runs) of the solutions. We observe that the models are sampling quite uniformly. More specifically, the ratio between the highest solution frequency and the lowest solution frequency is reduced to a factor of around 10, compared to a factor of 17,000 when using the random walk strategy alone.

4 ApproxCount algorithm

Since SampleSat produces efficient near-uniform sampling of the solution space, ApproxCount extends it to an approximate model counter based on the work of Jerrum *et al* [7]. The idea is to count the model of formula F, we first draw K samples from the solution space of F. (Note that a sample is a satisfying truth assignment.) The value of K is determined by the accuracy we want for our algorithm. If we consider a variable x_1 in F, and among the K samples, we denote the number of samples in which x_1 is assigned truth value *True* as $\#(x_1 = \text{True})$, and the number of samples in which x_1 is assigned truth value *False* as $\#(x_1 = \text{False})$. Assume the K samples are drawn from the uniform distribution over the models, and K is sufficiently large, then

$$\frac{\mathcal{M}(F \wedge x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{True})}{K}, \quad \text{and} \quad \frac{\mathcal{M}(F \wedge \neg x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{False})}{K}.$$

To ensure the stability of the algorithm, in each step we always pick truth value t such that $\#(x_1 = t) \geq \#(x_1 = \neg t)$. Without loss of generality, we assume t is *True*. The approximation above is equivalent to

$$\mathcal{M}(F) \approx \frac{K}{\#(x_1 = \text{True})} \cdot \mathcal{M}(F \wedge x_1).$$

$M_{x_1} = \frac{\mathcal{M}(F)}{\mathcal{M}(F \wedge x_1)}$, called multiplier of variable x_1, can be approximated by $\frac{K}{\#(x_1 = \text{True})}$, and formula $(F \wedge x_1)$ is simplified with unit propagation and the simplified sub-formula $F_{x_1 = \text{True}}$ can be counted by the above procedure recursively. The overall calculation is

$$\begin{aligned}\mathcal{M}(F) &= \frac{\mathcal{M}(F)}{\mathcal{M}(F_{x_1=t_1})} \cdot \frac{\mathcal{M}(F_{x_1=t_1})}{\mathcal{M}(F_{x_1=t_1,x_2=t_2})} \cdot \ \cdots \ \cdot \frac{\mathcal{M}(F_{x_1=t_1,x_2=t_2,\ldots,x_{n-1}=t_{n-1}})}{1} \\ &= M_{x_1} \cdot M_{x_2} \cdot \ \cdots \ \cdot M_{x_n},\end{aligned}$$

where $t_1, t_2, \cdots, t_{n-1}$ are chosen such that the multipliers are always no greater than 2.

The outline of the algorithm is given in Figure 5. We have experimented with 4 heuristics for **PickVar** in the algorithm:

– *pickbiased*: always pick the variable that maximizes $|\#(x = \text{True}) - \#(x = \text{False})|$. The variables first selected by this heuristic are likely backbone variables, and setting them to their right values may help simplify the whole formula.

```
**ApproxCount Algorithm**
**repeat**
        Draw $K$ samples from the solution space of $F$,
        $x :=$ choose a variable in $F$ by **PickVar**($F$)
        Among these $K$ samples,
        **if** $\#(x = \text{True}) > \#(x = \text{False})$
                $F :=$ Unitprop($F$, $x = \text{True}$)
                multiplier $M_x := K/\#(x = \text{True})$
        **else**
                $F :=$ Unitprop($F$, $x = \text{False}$)
                multiplier $M_x := K/\#(x = \text{False})$
**until** $F =$ **empty**
**output** product of all multipliers.
```

Fig. 5. Approximate Counts.

- *pickunbiased*: always pick the variable that minimizes $|\#(x = \text{True}) - \#(x = \text{False})|$. The variables first selected by this heuristic are likely don't-care variables, and setting them early may help the accuracy of counting.
- *pickrandom*: pick a variable uniformly at random.
- *pickbyorder*: first choose variable 1, and then variable 2, etc. In many structured problems, the variable order in the input formula often carries some domain information. For example, in graph coloring problems, the first variable may represent the first node is colored with the first color, etc. Following this order may help model counting. This heuristic is also designed for users who want to specify the order in which the variables should be set. To do so, users just need to rename the variables in input formula according their desired order of assignment.

We did experiments to compare the effectiveness of the first three heuristics. (The last one is fully user-definable and encoding dependent, and will require separate study.) Table 1 shows that *pickrandom* clearly dominates the other 2 heuristics. In all experiments that we will show in the next section, we used heuristic *pickrandom*.

Table 1. We rank heuristics *pickbiased*, *pickunbiased*, and *pickrandom* according to their accuracy in a suite of 100 formulas. The suite includes 50 random formulas and 50 structured formulas.

POSITION	*pickrandom*	*pickunbiased*	*pickbiased*
1ST PLACE	78	18	4
2ND PLACE	10	59	31
3RD PLACE	12	23	65

The run time of ApproxCount algorithm is upper-bounded by the product of the number of variables n, the number of solutions drawn in each iteration K, and the time needed to draw a solution c. Interesting and hard solution counting problems are usually under-constrained and have many solutions[1]. For these

[1] We will discuss more about this in Section 5.1.

problems, c does not increase drastically with the problem size, and ApproxCount has a polynomial run time with regard to the problem size. In the left pane of Figure 7, we show the run time of ApproxCount for a class of synthetic formulas.

5 Experimental Results

We tested our algorithm on formulas from a variety of domains. It is well-known in satisfiability testing that algorithms' performance depends heavily on the problem structures. This is also true for model counting. Algorithm that performs well in one domain may not work well in other domains.

5.1 Random Formulas

There has been much interest in counting the satisfying assignments of random 3CNF formulas. Random formulas are generated for different clause/variable (C/V) values. Unlike in satisfiability testing, where different algorithms all experience the peak of difficult at phase transition point around C/V = 4.26, the peak of difficulty for 3CNF model counting apparently shifts with the development of algorithms. Birnbaum and Lozinskii [2] report the peak of difficulty at C/V = 1.2 with their DPLL based CDP algorithm. Bayardo and Pehoushek [1] incorporate connected component analysis, and report the peak of difficulty at C/V = 1.5 with their DDP algorithm. Sang *et al* [12] add clauses learning and component caching to component analysis, and find the peak at the ratio of 1.8. Darwiche [3] compiles random 3CNF formulas to d-DNNFs, and also finds the peak at C/V = 1.8. Huang and Darwiche [6] use DPLL to compile these formulas to OBDDs, and find the peak at C/V = 2.0. The peak of difficulty is away from the peak of satisfiability test at 4.26 for these algorithms because instead of stopping at the first model, the algorithms need to visit every branch of the search tree with models. Formulas with less constraints have many more models, and make the counting harder.

For ApproxCount algorithm, counting models of formulas with the same number of variables and different ratios consumes almost the same amount of time[2]. Figure 6 shows the quality of approximation for different C/V ratios. The x-axis is the C/V ratio for random 3CNF formulas, and y-axis is the average error rate. If the approximate model count is a, and exact model count is t, the error rate is defined as $\frac{|a-t|}{\min(a,t)}$. 20 instances were generated at each ratio. From the figure, we see the error rates are quite low considering that approximation is NP-hard in the worst-case. We also observe that the error does seem relatively independent of the C/V ratio.

5.2 Application Domains

We also tested ApproxCount on structured formulas. Most of these formulas are taken from SATLIB [5]. The counting results are given in Table 2. In most of these domains, ApproxCount approximates the true model count within a factor of 2 or better. There are some domains, such as Boolean Vector, that are harder than others for ApproxCount.

[2] Actually higher ratio formulas have more clauses, and make ApproxCount spend slightly more time than lower ratio formulas, but the difference is very small.

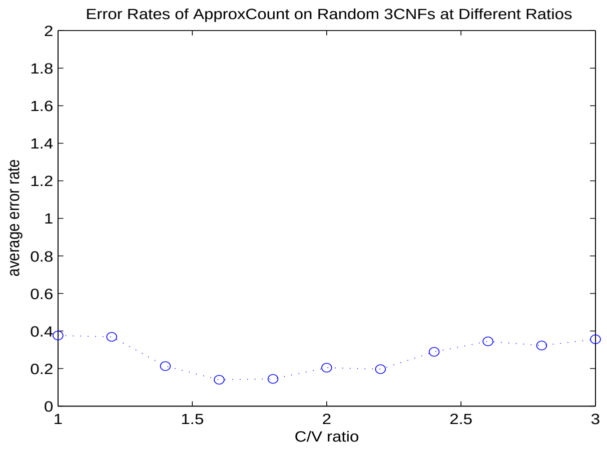

Fig. 6. Average error rates for ApproxCount on random 3CNF formulas with 75 variables. 1000 samples were drawn in each iteration. 20 instances were generated at each ratio. Exact model counts of the instances were calculated by Cachet [12].

5.3 Synthetic Domains

So far we have discussed instances that DPLL-based model counters can count exactly in order to evaluate the accuracy of our approximate model counter. However, ApproxCount can also provide good estimates for formulas that DPLL-based counters cannot count in reasonable time and memory limits. To show this, we need to design a class of formulas that we know the exact model counts by other means.

For this reason, we encode the following combinatorial problem to Boolean logic: suppose there are n different items, and you want to choose from the n items a list (order matters) of m different items ($m \leq n$). Let $P(n, m)$ represent the number of different lists you can construct. The value of $P(n, m)$ is given by $P(n, m) = \frac{n!}{(n-m)!}$.

The encoding to propositional logic works as follows. For each position i, and each item j, we use a Boolean variable $x_{i,j}$ to represent "whether position i will hold item j". We have the following three kinds of clauses:

– "each position holds at most one item": this translates into $mn(n-1)/2$ clauses
$$\neg x_{i,j_1} \vee \neg x_{i,j_2}, \quad \text{for any } i, j_1 > j_2;$$
– "each position holds at least one item": this translates into m clauses
$$\vee_{j=1}^{n} x_{i,j}, \quad \text{for any } i;$$
– "different positions hold different items": this translates into $nm(m-1)/2$ clauses
$$\neg x_{i_1,j} \vee \neg x_{i_2,j}, \quad \text{for any } j, i_1 > i_2.$$

Table 2. ApproxCount results in application domains.

All Interval Series				
instance	#var	#clauses	#models	ApproxCount result
ais6	61	581	24	24
ais8	113	1520	40	40

Circuit Fault Analysis				
instance	#var	#clauses	#models	ApproxCount result
ssa7552-158	1363	3064	25×10^{30}	7×10^{30}
ssa7552-159	1363	3032	7×10^{33}	3×10^{33}

Graph Coloring				
instance	#var	#clauses	#models	ApproxCount result
flat30-3	90	300	1968	2422
flat30-4	90	300	720	985
flat30-5	90	300	1362	1338

Boolean Vector				
instance	#var	#clauses	#models	ApproxCount result
2bitcomp_5	125	310	9.8×10^{15}	3.6×10^{15}
2bitcomp_6	252	766	21×10^{28}	3.8×10^{28}

Logistics				
instance	#var	#clauses	#models	ApproxCount result
prob004-log-a	1790	18026	2.6×10^{16}	1.4×10^{16}

Bounded Model Checking				
instance	#var	#clauses	#models	ApproxCount result
dp02s02.shuffled	319	683	1.5×10^{25}	1.2×10^{25}

Therefore, each instance of the problem is encoded to a Boolean formula with mn variables and $(mn(m+n-2)/2+m)$ clauses. These formulas seem hard for DPLL-based counters. As shown in the left pane of Figure 7, with the increase of length of the list, the run time of both Relsat [1] and Cachet [12] grows exponentially. The run time of ApproxCount is polynomially related to the problem size. In the right pane, we see that the estimates produced by ApproxCount are very close to the theoretical results.

Table 3. ApproxCount results on $P(n, m)$. The last column gives the average error in each step that actually contributes to the final error.

n	m	#var	#clauses	#models	ApproxCount result	error per step
20	10	200	2810	6.7×10^{11}	7.2×10^{11}	0.05%
20	15	300	4965	2.0×10^{16}	1.6×10^{16}	0.09%
20	20	400	7620	2.4×10^{18}	2.8×10^{18}	0.04%
25	10	250	4135	1.2×10^{13}	1.1×10^{13}	0.04%
25	15	375	7140	4.3×10^{18}	4.8×10^{18}	0.03%
25	20	500	10770	1.3×10^{23}	1.4×10^{23}	0.02%
30	10	300	5710	10.9×10^{13}	9.1×10^{13}	0.07%
30	15	450	9690	2.0×10^{20}	2.1×10^{20}	0.01%
30	20	600	14420	7.3×10^{25}	5.9×10^{25}	0.04%

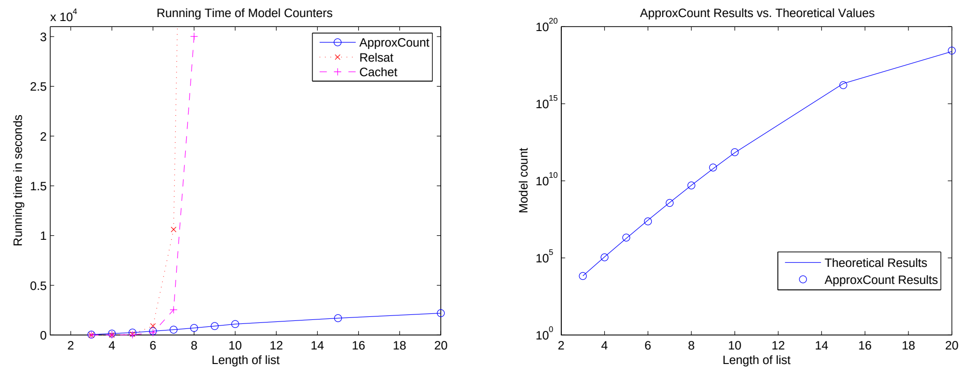

Fig. 7. Performance of model counters for combinatorial problem $P(20, m)$. In both figures, x-axes represent the value of m.

We have experimented with larger formulas in the class than those given in Figure 7 to see how the algorithm scales. The results are given in Table 3. From the table, we see the algorithm scales well on this class of formulas. Because ApproxCount calculates the total count by the product of estimated multipliers of each variable, the error at each step could potentially accumulate to a large overall error. To understand why this does not happen, we calculate the error rate of ApproxCount at each step of a small formula $P(20, 4)$ using Cachet, and find the average error rate (as defined in Section 5.1) of each estimated multiplier is 0.71%, with a standard deviation of 0.48%. However, in about 50% of the steps ApproxCount over-estimates the multipliers, and in the other 50% of the steps it under-estimates the multipliers. Overall these errors cancel out each other for the most part, and result in an overall error rate of around 5%. If we distribute this error to the 80 steps, each step only contributes 0.06% to the final error, and other part is canceled by the opposite errors at other steps. For larger formulas, however, we can no longer calculate error rate at each step. In the last column in Table 3, we calculate the average error rate at each step that contributes to the final error. This error rate is consistently low across all sizes of formulas in the table.

6 Dealing with Real Numbers in Probabilistic Reasoning

In many probabilistic reasoning models, such as in Bayesian networks, probabilities are represented by real numbers, therefore the reasoning tasks can often be converted naturally to weighted model counting of Boolean formulas [12][3]. In weighted model counting, each variable a is assigned a weight $w_a \in [0, 1]$, and its negation $\neg a$ is assigned weight $1 - w_a$. $w(a = t)$ is defined as w_a when t is *True*, and $1 - w_a$ when t is *False*. The weight of a model is the product of the weights

[3] Discussion about the conversion is available at http://www.cs.washington.edu /homes/kautz/talks/counting-sat04.ppt.

of its literals. The weighted model count of a formula F, noted as $\mathcal{M}_w(F)$, is the sum of the weights of all of its models.

Unweighted counting we discussed in previous sections can be considered as a special case of weighted counting where each variable has a weight 0.5. We can use pure propositional logic to encode these real number weights. For example, if a variable a has a weight 0.375 we can encode a as $(a_1 \wedge a_2) \vee (a_3 \wedge a_4 \wedge a_5)$, and use unweighted model counter to count the model. The weighted model count can be calculated as the model count of the encoded formula divided by 2^n, where n is the number of variables. However, this approach introduces many new variables, and makes the counting inefficient.

ApproxCount algorithm can be easily modified to calculate weighted model count. Because

$$\begin{aligned}
&\mathcal{M}_w(F) \\
&= \Big(\frac{\mathcal{M}_w(F)}{\mathcal{M}_w(F_{x_1=t_1}) \cdot w(x_1 = t_1)} w(x_1 = t_1)\Big) \\
&\quad \cdot \Big(\frac{\mathcal{M}_w(F_{x_1=t_1})}{\mathcal{M}_w(F_{x_1=t_1,x_2=t_2}) \cdot w(x_2 = t_2)} \cdot w(x_2 = t_2)\Big) \cdot \\
&\quad \cdots \cdot \Big(\frac{\mathcal{M}_w(F_{x_1=t_1,x_2=t_2,\ldots,x_{n-1}=t_{n-1}})}{1 \cdot w(x_n = t_n)} \cdot w(x_n = t_n)\Big) \\
&= \big(M_{x_1} \cdot w(x_1 = t_1)\big) \cdot \big(M_{x_2} \cdot w(x_2 = t_2)\big) \cdot \ \cdots \ \cdot \big(M_{x_n} \cdot w(x_n = t_n)\big).
\end{aligned}$$

At each step, the multiplier is estimated by the sum of weights of all samples divided by the sum of weights of samples that assign truth value t to the variable in consideration, where t is chosen such that the multiplier is no greater than 2 to maintain the stability of the estimate.

```
Weighted ApproxCount Algorithm
product := 1;
repeat
    Draw $K$ samples from the solution space of $F$,
    $x$ := choose a variable in $F$ by PickVar($F$)
    Among these $K$ samples,
    if $\#_w(x = \text{True}) > \#_w(x = \text{False})$
        $F$ := UnitProp($F$, $x$ = True)
        multiplier $M_x := w(K)/\#_w(x = \text{True})$
        product := product * multiplier * $w(x = \text{True})$
    else
        $F$ := UnitProp($F$, $x$ = False)
        multiplier $M_x := w(K)/\#_w(x = \text{False})$
        product := product * multiplier * $w(x = \text{False})$
until $F$ = empty
output product.
```

Fig. 8. Approximate Weighted Model Counts.

Figure 8 gives the modified ApproxCount algorithm. In the algorithm, $w(K)$ represents the sum of weights of the K samples drawn, and $\#_w(x = \text{True/False})$

represents sum of weights of samples that assign x to *True*/*False* among the K samples drawn.

7 Conclusion

We have presented ApproxCount algorithm that approximates the model count of a formula in propositional logic. We have shown that ApproxCount generates good estimates for formulas in several domains. The approach extends the range of formulas whose models can be counted approximately. ApproxCount proceeds incrementally, setting one variable at a time and computing a multiplier at each step. Most interestingly, our work suggests that the individual errors in the multipliers have a tendency to cancel out, thereby making the approach a very promising one. We also proposed modifications of the algorithm to deal with real numbers in probabilistic reasoning models.

References

1. R. J. Bayardo, Jr. and J. D. Pehoushek. Counting Models Using Connected Components. In *Proc. AAAI-00*, 2000
2. E. Birnbaum and E. L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research*, 10:457-477, 1999.
3. A. Darwiche. A Compiler for Deterministic, Decomposable Negation Normal Form. In *Proc. AAAI-02*, 2002.
4. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394-397, 1962.
5. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *I. P. Gent, H. v.Maaren, T. Walsh, editors, SAT 2000* , pp.283-292, IOS Press, 2000. SATLIB is available online at www.satlib.org.
6. J. Huang and A. Darwiche. Using DPLL for Efficient OBDD Construction. In *Proc. SAT-04*, 2004
7. M. R. Jerrum, L. G. Valiant, V. V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science* 43, 169-188, 1986.
8. H. Kautz and B. Selman. Planning as Satisfiability. In *Proceedings ECAI-92*, 1992.
9. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering a Highly Efficient SAT Solver. *38th Design Autom. Conference (DAC 01)*, 2001.
10. C. H. Papadimitriou. On Selecting a Satisfying Truth Assignment. In *Proceedings of the Conference on the Foundations of Computer Science*, pages 163-169, 1991.
11. D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence* 82, 273-302, 1996.
12. T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proc. SAT-04*, 2004.
13. B. Selman, H. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. *2nd DIMACS Challenge on Cliques, Coloring and Satisfiability*, 1994.
14. S. Toba. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM Journal on Computing*, Vol. 20, No. 5, 865-877, 1991.
15. L. G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, Vol. 8, No. 3, 410-421, 1979.
16. M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, Vol. 35, No. 2, pp. 73-106, 2003.
17. W. Wei, J. Erenrich, and B. Selman. Towards Efficient Sampling: Exploiting Random Walk Strategies. In *Proc. AAAI-04*, 2004.
18. W. Wei and B. Selman. Accelerating Random Walks. In *Proc. CP-02*, 2002.
19. SAT Competition. http://www.satcompetition.org