Structural Bit-vector Model Counting

Seonmo Kim and Stephen McCamant

University of Minnesota, Minneapolis, Minnesota, U.S.A.
{smkim, mccamant}@cs.umn.edu

Abstract

Various approximate model counting techniques have been proposed and are used in many applications such as probabilistic inference and quantitative information-flow security. The hashing-based technique is a well-known approach and can be more scalable than exact model counting techniques. However, its performance is highly dependent on the performance of a decision procedure (SAT or SMT solver) and adding numerous hashing constraints to a formula might cause a solver to perform poorly.

We propose a model counting technique which computes lower and upper bounds of the number of solutions to an SMT formula by analyzing the structure of the formula, which means this approach does not rely on a decision procedure. Our algorithm runs in polynomial time and gives firm lower and upper bounds unlike other approximate model counters that compute probabilistic bounds. We compare our algorithm with state-of-the-art model counters and our experiments show that our approach is faster than others and provides a trade-off between computational effort and the precision of results.

1 Introduction

Model counting is a quantitative generalization of the satisfiability problem that asks for the number of satisfying assignments for a formula. Some of the many applications of model counting include combinatorics, safety analysis, probabilistic inference, and quantitative information-flow analysis of software. Model counting is also closely connected to uniform random sampling of formula solutions. However, exact model counting even just for quantifier-free Boolean formulas (#SAT) is complete for the complexity class #P which is believed to be intractable. Approximations to an exact model count are useful in many applications, so the difficulty of exact model counting motivates exploring approximation algorithms.

The best-known approaches in recent research use hashing to reduce approximate model counting to an adaptive sequence of satisfiability queries covering subsets of the solution space. These algorithms provide probabilistic bounds on the accuracy of their approximation, and they can take advantage of advances in satisfiability solving (both SAT and SMT). However these approaches still have limited scalability in practice: they require a large number of satisfiability queries to achieve tight bounds, and hashing can make individual queries much more expensive, given the complex interactions between hashing constraints and solver optimizations.

Also valuable but less developed are approximation algorithms which achieve guaranteed efficiency and sound bounds, at the expense of not providing an accuracy guarantee. This trade-off can be achieved using algorithms similar to ones used in static program analysis that derive lower and/or upper bounds following the syntactic structure of a formula rather than using semantic decision procedure queries. We refer to these as "structural" approximate model counting algorithms.

Previous structural model counting algorithms have been specialized for narrow domains, or have been built into larger systems in ways that are not easily reusable. In this work, we build a new structural approximate model counting tool for quantifier-free SMT formulas over the theory of bit vectors (SMT-LIB QF_BV), one of the most common theories used to model

François Bobot, Tjark Weber (eds.); SMT 2020, pp. 26–36
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2854/short2.pdf

bounded arithmetic and software semantics. We also provide a commonly useful generalization known as projected model counting, in which a user can specify a subset of the variables in a formula over which a model should be counted. Our tool uses algorithms which build on the partial description of a previous closed-source tool [16], but we needed to develop new structural rules for cases that were missing or restricted in previous work. We extend the algorithm to cover a more complete set of bit-vector operators, and to use both the signed and unsigned orderings of bit vectors. However our current implementation, like Martin's [16], is limited to conjunctions and not arbitrary Boolean combinations of bit-vector relations. This matches the needs, for instance, of typical single-path symbolic execution, where a path condition is a conjunction of branch conditions, each of which is an equality or inequality over numeric (e.g. bit-vector) terms.

We have built a standalone tool, SMC, that operates on input in the standard SMT-LIB2 format, and which we have open-source. We have checked the correctness of the propagation rules for each type of bit-vector expression via bounded exhaustive testing. We empirically compare SMC's performance with representative state-of-the-art exact and approximate model counting tools. The results show that SMC's performance scales much better than these other tools, and that the sound upper and lower model-count bounds that it provides are often usefully accurate.

2 Related Work

We categorize previous model-counting tools based on exact approaches, hashing-based approximation, and structural approximation.

Exact model counting Exact model counting for propositional formulas (#SAT) typically uses techniques related to SAT solving: intuitively, a #SAT algorithm must traverse the entire search space even for a satisfiable formula, instead of stopping at the first satisfying assignment as a SAT solver does. The counting process is optimized by caching information about the number of solutions for sub-formulas that appear repeatedly in the total. Relsat [4], Cachet [19], and sharpSAT [24] are some examples of tools based on varieties of caching. dSharp [18] builds on sharpSAT and also computes a conversion of Boolean formula to a restricted normal form known as d-DNNF (deterministic, decomposable negation normal form) which facilitates other operations. dSharp_p [2] further builds on dSharp to support projected model counting; it is the exact model counting tool we compare with in our evaluation below. Two other notable more recent systems are countAntom [5], which uses a parallel algorithm, and Ganak [20], which uses probabilistic hashing to save space, but still produces an exact model count with a guaranteed minimum probability.

Hashing-based model counting The core principle that enables hashing-based approximation was first deployed for theoretical results, notably by Sipser [21], Stockmeyer [23], and Valiant and Vazirani [25]. It was first applied to build model counting tools by Gomes et al. [12], who gave an approach that probablistically checked a hypothesized lower or upper bound. Later systems achieved complete automation by designing outer-loop algorithms to search over model count hypotheses. The best known tool of this kind is ApproxMC [7] and its successors including ApproxMC2 [8], and BIRD/ApproxMC3 [22], and a projecting variant ApproxMC-p [14]. SearchMC [13] reduces the number of queries needed by keeping a statistical estimation of the model count, and also implements a mode that can apply directly wrap an SMT solver (tools

```
(declare-fun x () (_ BitVec 8))
(declare-fun y () (_ BitVec 8))
(assert (= y (bvadd (bvand x 15) 4)))
```

Figure 1: Simple Example

designed solely for #SAT require eager bit-blasting). We take SearchMC as the representative of hashing-based approximate model counters in our evaluation.

Structural model counting Structural approaches to approximate model counting that can provide non-probabilistic bounds (but not guaranteed precision) have seen relatively less tool development. The most direct predecessor of our work in this paper is the FSCB (Fast Solution Count Bounder) algorithm of Martin [16], which as far as we are aware was applied only as part of symbolic execution system to estimate the amount of information revealed by bug reports [6]. Martin's implementation was not available, but we used the description in his technical report as a starting point for the system we developed, as described in more detail in Section 3. Other previous structural model-counting systems have been specialized for other domains. Luu et al. [15] build a model counter for string constraints such as arise in symbolic execution of high-level languages like JavaScript; their approach is based on generating functions. Aydin et al.'s MT-ABC [1] uses a combination of structural and automaton-based algorithms for constraints that can include a mix of strings and linear arithmetic. Meng and Smith's two-bit-pattern technique [17] is a hybrid of structural and exact counting approaches to #SAT. It structurally over-approximates the model count by determining, for every pair of bits in a formula, what values they can have in isolation (using many small satisfiability queries). Then the combination of these constraints is a 2CNF formula, which the authors found could be model-counted efficiently in practice (though general 2CNF model counting is still #P-hard).

3 Structural Model Counter

The Structural Model Counter (SMC[1]) takes as input an SMT-LIB2 formula which consists of variables and assertions. It outputs lower and upper bounds of satisfying assignments that make a given formula true. First, we describe how this structural model counting technique works with a simple example in Fig 1. Since `x` and `y` are 8-bit variables, the model counts of two variables are both 256 when they are declared. Then we parse an assertion to analyze the structure of the formula. `(bvand x 15)` has the minimum 0 and the maximum 15. This generates 16 distinct values. Adding 4 makes the minimum 4, the maximum 19 and still 16 distinct values. This is equal to `y` thus `y` has 16 distinct values. The model count of this formula is 256 since `x` has 256 distinct values and `x` determines `y`. This shows a simple process of the structure model counting. In this section, we describe this algorithm in detail, its correctness, some current limitations and differences from the previous work.

[1]The source code and benchmarks are available at: https://github.com/seonmokim/smc

3.1 Algorithm

The main algorithm is mostly inspired by Martin's FSCB (Fast Solution Count Bounder) algorithm [16]. He proposed this idea to be a fast algorithm that can handle complex expressions such as standard arithmetic or bit shifts. However, FSCB does not consider signed data types and the source code is not available. We extend FSCB to handle both unsigned and signed data types and cover more operators including signed division, signed less-than, signed greater-than and so on. Also, we improve the idea to compute tighter bounds and verify its correctness using small-sized unit tests exhaustively. In this section, we describe two steps of our algorithm. It first computes the bounds for each individual assertion, and then merges them for the model count of a given formula.

3.1.1 Per-Assertion Bounds and Analysis

When a variable is declared or an expression is generated, we create a corresponding node to represent this variable or expression. Each node contains elements as [`ul, uh, sl, sh, lc, hc, hom, vars`]. Since we only consider bit vectors in SMT-LIB2 format, we compute both cases, unsigned and signed representations, in a node. `ul`, `uh`, `sl` and `sh` are the unsigned minimum (low), the unsigned maximum (high), the signed minimum and the signed maximum of the node, respectively. `lc` and `hc` are the low cardinality (lower bound), the high cardinality (upper bound) of the node, respectively. We select the bounds which have a shorter interval between the unsigned and signed bounds since both the bounds are sound and tighter bounds give a better precision. `hom` is a flag such that a node is homogeneous only if every image has the same number of preimages. For example, `(bvand x 15)` is homogeneous since it generates 16 distinct values and each value (image) has the same number of preimages (16 cases if `x` is 8-bit). This flag is useful when computing bounds more precisely with a constant value. For example, let us assume that we have `(= (bvand x 15) 0)` and `x` is 8-bit. `(bvand x 15)` should be zero to satisfy this assertion and one value of `(bvand x 15)` maps to 16 preimages of `x`. Therefore, this can be computed easily as `x` has exactly 16 distinct values to satisfy this assertion. `vars` is a set of variables which presents in an assertion. As we described above, we represent a signed and unsigned values in a single node. One reason that we need both representations is that we want to handle signed operators such as signed division, signed less-than, signed greater-than and so on. Another reason is the ways of computing signed values and unsigned values are different. Therefore, we maintain both representations in every operation and give a warning whenever SMC sees a conflict case.

SMC first replaces each constant or variable from an expression with the corresponding node. If it is a constant `c`, the node can be represented as [`c, c, c, c, 1, 1, true,` {}]. Note that if c is a binary or hexadecimal number, we convert the number using the two's complement. The next step is that SMC breaks down an expression tree and generates a node based on the expression rules starting from the sub-expressions. SMC determines how to compute the node values from the expression rules for each supported operation. We can extend SMC to support additional operations by adding new operation rules. We only show a subset of the operation rules here.

Pseudocode for `bvadd` is shown in Fig 2. When the tool reaches a `bvadd` operation, it first checks whether the variables are both constant values. This checking applies to other operations as well and we can easily compute all the values if they are both constant values. Next we check that the answer might be an arithmetic positive or negative overflow. If this occurs, we set `ul` and `uh` as the minimum and the maximum of the variable, respectively, based on its bit-width unless the variables are both constants. This applies to `sl` and `sh` to be the

```
(bvadd f g)
  if isConstant(f) and isConstant(g):
    ul = uh = (f.ul + g.ul) % 2^f.width
    sl = sh = (f.sl + g.sl) % 2^f.width
    return ul, uh, sl, sh, 1, 1, true, [f.vars ∪ g.vars]

  if f.uh + g.uh > umax:
    ul = umin
    uh = umax
  else:
    ul = f.ul + g.ul
    uh = f.uh + g.uh

  if f.sl + g.sl < smin or f.sh + g.sh > smax:
    sl = smin
    sh = smax
  else:
    sl = f.sl + g.sl
    sh = f.sh + g.sh

  lc = min(f.lc+g.lc-1, abs(uh-ul), abs(sh-sl))
  if isCommon(f,g):
    lc = 1
  hc = min(f.hc * g.hc, abs(uh-ul), abs(sh-sl))
  hom = (f.hom and isConstant(g))
     or (isConstant(f) and g.hom)
     or (not isCommon(f,g) and isPerm(f) and isPerm(g))

  return ul, uh, sl, sh, lc, hc, hom, [f.vars ∪ g.vars]
```

Figure 2: The operation rule of bvadd

negative minimum and positive maximum. `lc` and `hc` can be computed as the sum of two variable's minimum cardinality minus 1. In order to generate the smallest set of adding two sets, one value has to be generated in multiple ways. For example, adding {1, 2, 3} and {2, 3} makes {3, 4, 5, 6} which has 4 elements. If the function has variables in common (for example, `(x&1)+(¬x&1)`), the minimum cardinality could be 1. The maximum cardinality can be computed as the multiplication of each variable's maximum cardinality. Note that the cardinalities must be less than the distance between its high value and low value. The addition is homogeneous if one variable is homogeneous and another is a constant or they do not have variables in common and both are permutations. A variable is a permutation if it is homogeneous and unconstrained.

Let us go back to the example in Fig 1 and proceed with the SMC algorithm. When `x` and `y` are declared, two corresponding nodes are generated. Initial values of `x` would be [`0, 255, -128, 127, 256, 256, true,` {`x`}] and initial values of `y` would be the same except `vars=`{`y`}. If we have the equation `(= y (bvadd (bvand x 15) 4))`, we parse the equation and generate a node from the expression. `(bvand x 15)` returns a node with

```
(= f g)
  ...
  leq = max(f.ul, g.ul)
  heq = min(f.uh, g.uh)
  minlhit = f.lc - max(f.uh - heq, leq - f.ul)
  minlhit = min(minlhit, heq - leq + 1)
  minrhit = g.lc - max(g.uh - heq, leq - g.ul)
  minrhit = min(minrhit, heq - leq + 1)
  inter = max(1, minlhit + minrhit - (heq - leq + 1))
  ic = 2^(f.width + g.width)
  if f.hom:
    ld_f = 2^f.width / f.hc
    hd_f = 2^f.width / f.lc
  else:
    ld_f = 1
    hd_f = 2^f.width - f.lc + 1
  if g.hom:
    ld_g = 2^g.width / g.hc
    hd_g = 2^g.width / g.lc
  else:
    ld_g = 1
    hd_g = 2^g.width - g.lc + 1
  lb = max(1, min(inter * ld_f * ld_g, ic))
  hb = min(inter * hd_f * hd_g, ic)
  ...
```

Figure 3: The operation rule of =

[0, 15, 0, 15, 16, 16, true, {x}] and then (bvadd (bvand x 15) 4) returns a node with [4, 19, 4, 19, 16, 16, true, {x}]. When there is an equality or inequality check in an assertion, SMC computes a lower bounds and upper bound of variables in the assertion. We denote lb and hb which are the lower and upper bounds of a set of variables, respectively. Fig 3 shows the crucial part to compute the bounds.

This equal rule computes the cardinalities of the left-hand side and the right-hand side and then the cardinality of the intersection between the left-hand side and the right-hand. Depending on the homogeneity of the variable, we compute the low and high density of the variable so we can compute the lower bound and upper bound. We also apply the same procedure to signed values and select the bounds that has a smaller interval. In this example, SMC computes the lower bound as 256 and the upper bound as 256, which is an accurate answer.

3.1.2 Combining Bounds

The second step is to combine per-assertion bounds. For each assertion with comparison operators such as equal, greater-than or less-than, we compute the lower and upper bounds on the number of solutions to variables in the assertion. Recall that lb and hb are the bounds of a set of variables. We use vars to denote the set of variables in an assertion. Given two

```
mergeBounds(Bounds a, Bounds b):
    vars = a.vars ∪ b.vars
    inter_vars = a.vars ∩ b.vars
    width = inter_vars.width
    inter_max = min(2^width, math.gcd(a.ha, b.ha))
    inter_min = min(2^width, math.gcd(a.la, b.la))
    if a.vars and b.vars are in common:
        if set(a.vars) == set(b.vars):
            la = min(a.la, b.la)
            ha = min(a.ha, b.ha)
        elif a.vars ⊂ b.vars:
            la = b.la
            ha = b.ha
        elif b.vars ⊂ a.vars:
            la = a.la
            ha = a.ha
        else:
            ula = a.ula * b.ula / inter_min
            uha = a.uha * b.uha / inter_max
    else:
        la = a.la * b.la
        ha = a.ha * b.ha

    return Bounds(la, ha, vars)
```

Figure 4: The operation rule of mergeBounds

such bounds, we can merge per-assertion bounds into bounds that apply to both equations together. We recursively merge the bounds pairwise until the bound represent all the variables in the system. For example, if two bounds are independent, we can simply multiply each lower bound and upper bound. We present pseudocode for merging bounds in Figure 4. We first check whether two bounds are independent or not. If they are independent, we can just simply multiply two bounds. If they have variables in common, then we consider four cases: they are identical, one is subset of another (or vice versa) and they are overlapping sets. We compute the bounds conservatively based on the case. The bounds can be more precise based on the order of merging such as merging similar variables first. This needs an extra running time and we left this for the future work.

3.2 Differences between SMC and FSCB

Here we briefly describe the differences between SMC and FSCB. First, we cover a more complete set of bit-vector operators for both the signed and unsigned orderings of bit vectors. Table 1 shows which bit-vector operators are supported by FSCB and SMC.

Also, we added more edge cases in operators and fixed some operators to compute a better precision. For example, `bvadd` in FSCB computes the low cardinality as `max(f.lc, g.lc)` but we empirically find out that `f.lc+g.lc-1` gives a better result.

FSCB	`bvadd`, `bvsub`, `bvmul`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `=`, `distinct`, `ult`, `ugt`
SMC	`bvadd`, `bvsub`, `bvmul`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `=`, `distinct`, `ult`, `ugt`, `ule`, `uge`, `slt`, `sgt`, `sle`, `sge`, `bvudiv`, `bvsdiv`, `concat`, `extract`, `sign_extend`

Table 1: Bit-vector operators supported by SMC and FSCB

3.3 Correctness

We have tested the correctness of the operation rules of bit-vector expression using bounded exhaustive checking. We set each variable to be a bit-vector of width 4 and executed each operator with the power set of each variable. We checked whether `ul`, `uh`, `sl`, `sh`, `lc` and `hc` from our operation rules are sound. We verify unary operators and binary operators. First, we compute `ul`, `uh`, `sl`, `sh`, `lc` and `hc` based on our operation rules. We generate the power set of one variable which has 2^{16} sets and execute an operation which generates 2^{32} sets for binary operators. The same procedure is applied to unary operators. We verify that (1)the unsigned smallest value of each set is greater than `ul`, (2)the unsigned largest value of each set is less than `uh`, (3)the signed smallest value of each set is greater than `sl`, (4)the signed largest value of each set is less than `sh` and (5)the number of elements of each set is less than `hc` and greater than `lc`. However, we have not checked the correctness of merging bounds due to the state explosion problem since we need to test the case where bounds contain multiple variables.

3.4 Limitation

In this section, we discuss the limitations of our current implementation. We have extended FSCB to support more operators in SMT-LIB2 format. However, some of SMT-LIB2 operators are not supported in SMC and we are currently working on handling the whole SMT-LIB2 format standard. Also, some operators compute the cardinalities very conservatively due to the limitation of the node representation, hence those operators lead to less precise results (loose bounds). This is for the future work to have more coverage of SMT-LIB2 format standard and design more precise rules.

Lastly, the order of assertions and merging bounds affects precision. For example, let say we have two variable v_1 and v_2 and one constant c. If we have assertions $(= v_1\ v_2)$ and $(= v_2\ c)$, the results are computed differently depending on which assertion is processed first. For example, $(= v_1\ v_2)$ computes the bounds $[256, 256]$ and $(= v_2\ c)$ computes the bounds $[1, 1]$ if all the variables are 8-bit. If we merge the two bounds, the final bound would be $[256, 256]$. But if we flip the order which $(= v_2\ c)$ computes the bounds first, then $(= v_1\ v_2)$ computes the bounds $[1, 1]$ since we know that v_2 is a constant value this time. This can be resolved by recursively computing the bounds until the bounds do not change but we do not have any proof about its time complexity. This means the bounds can be more precise by merging the per-assertion bounds in a better order. We believe finding more efficient way to merge bounds will improve the performance and this is for the future work.

4 Experimental Results

In this section, we show our experimental results and all our experiments were performed on a machine with an Intel Core i7 3.40Ghz CPU and 16GB memory. We implement our algorithm with Python and use our own SMT-LIB2 parser. Our algorithm supports SMT-LIB2 format [3]. We compare our algorithm with state-of-the-art model counters: SearchMC [13] and

dSharp_p [2]. SearchMC is an approximate model counter using XOR hashing constraints to estimate a lower bound and upper bound of the model count. It is a randomized algorithm and gives a desired level of distance between a lower bound and upper bound with a probability of at least 0.6. In this experiment, we ran SearchMC 10 times for each benchmark until it gave the first bounds with a probability of at least 0.6 and computed the average of the results. dSharp_p is an exact model counter and is implemented on top of dSharp [18] to support projection. We collected various SMT_BV benchmarks from previous works [10, 11]. Here we show partial results of some representative benchmarks. Table 2 shows a comparison on performance and approximation. The second column shows the number of bits we want to count over. Since dSharp_p is an exact model counter, we take $\log_2$ of the answer which shows in the third column and the running times (in seconds) of dSharp_p are shown in the fourth column. We also show the bounds (log base 2) computed from SearchMC and SMC following with their running times. dSharp_p performs well on a small-sized problems and its performance decreases as the size and the complexity of formula increases. In these experiments, SearchMC used the state-of-the-art SMT(BV) solver Z3 [9] and its performance was highly dependent on the performance of the solver.

SMC shows a good precision on some benchmarks if the structure of the formula is well-organized. However, it gave very loose bounds on some benchmarks which SMC was not able to analyze the formula well. For example, 5_10_1 and 5_20_1 are the volume computation problems of convex bodies and consist of a number of inequality constraints. This type of problems show very loose bounds since SMC computes the bounds very conservatively on inequality constraints. The main benefit of SMC is the running times. This shows that our approach is faster than others and it is a trade-off chosen between computational effort and the precision of results in some benchmarks.

Hashing-based model counting techniques like SearchMC rely on prior hypotheses to produce more useful results and start an initial hypothesis from zero knowledge. The initial hypothesis for SearchMC is a uniform distribution over 0 to the maximum bit-width of the output bit-vector. If we gather results from SMC and use them as the initial hypothesis for SearchMC, SearchMC is able to give a desired answer faster. For example, in order to solve coloring_4 the initial hypothesis for SearchMC is a uniform distribution over 0 to 32. If the initial hypothesis for SearchMC is a uniform distribution over 29.61 to 32 which is computed by SMC, SearchMC needs a smaller number of queries to find a desired result.

Table 3 shows the performance of the combination of SMC and SearchMC. The results for plain SearchMC are equivalent to the results in Table 2. We also measured the average number of iterations (loops) in SearchMC and the results show that the number of iterations was decreased when we used SMC and SearchMC together. Since SMC already computed tight bounds on gettoPath1 and calDate_10, we did not run SearchMC on the benchmarks. This experimental results show that using SMC as a preprocessor of SearchMC gives the performance benefit up to 1.36x speedup.

5 Conclusion

We propose a structural approximate model counting algorithm, SMC, to compute the lower and upper bound of solutions to a given SMT formula. This is a fast polynomial algorithm compared to other state-of-the-art approximate model counters and it runs in $O(n+m)$ where n is the number of variables and m is the number of assertions. We extend the FSCB algorithm to cover a more complete set of SMT-LIB2 standard operators and to use both the signed and unsigned representations of bit vectors. Our evaluation results illustrate that our technique is

		dSharp_p		SearchMC		SMC	
Benchmarks	#Bits	$\log_2(MC)$	t(s)	Bounds	t(s)	Bounds	t(s)
coloring_4	32	30.75	0.82	[30.15 31.05]	1088.52	[29.61 32.00]	0.001
FINDpath1	32	21.96	0.09	[20.9 22.17]	3.99	[4.00 32.00]	0.001
getopPath1	8	7.92	0.003	[7.50 8.25]	0.21	[7.92 7.94]	0.001
queue	16	6.39	0.006	[5.66 6.97]	0.17	[4.62 10.78]	0.001
calDate_10	36	16.95	0.002	[16.13 17.49]	0.30	[16.95 16.95]	0.001
5_10_1	160	12.02	123.12	[11.34 12.31]	20.73	[7.24 24.77]	0.003
5_20_1	160	8.44	351.57	[7.691 8.88]	29.6	[7.24 24.77]	0.004

Table 2: Comparison Result

		SearchMC			SMC+SearchMC		
Benchmarks	#Bits	Bounds	t(s)	#loops	Bounds	t(s)	#loops
coloring_4	32	[30.15 31.05]	1088.52	6.3	[30.16 31.53]	830.88	4.9
FINDpath1	32	[20.9 22.17]	3.99	8.2	[21.24 22.42]	3.55	7.7
getopPath1	8	[7.50 8.25]	0.21	4	-	-	-
queue	16	[5.66 6.97]	0.17	6.1	[5.60 6.78]	0.15	4.8
calDate_10	36	[16.13 17.49]	0.30	8.3	-	-	-
5_10_1	160	[11.34 12.31]	20.73	8.9	[11.2 12.44]	18.3	6.4
5_20_1	160	[7.691 8.88]	29.62	8.4	[7.42 8.85]	21.83	5.3

Table 3: Combination of SMC and SearchMC

most beneficial when time performance is a tight requirement.

Acknowledgments

We would like to thank the anonymous reviewers for suggestions which have helped us to improve our system and the paper's presentation. The research described in this paper has been supported in part by the National Science Foundation under grant 1526319 and by the Office of Naval Research under grant N00014-19-1-2541.

References

[1] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 400–410, 2018.

[2] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In *Theory and Applications of Satisfiability Testing*, 2015.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.

[4] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI*, pages 203–208, 1997.

[5] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-Faire caching for parallel #SAT solving. In *Proceedings of SAT*, pages 46–61, 2015.

[6] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of ASPLOS*, pages 319–328, 2008.

[7] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proceedings of CP*, volume 8124, pages 200–216, 2013.

[8] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Improving approximate counting for probabilistic inference: From linear to logarithmic SAT solver calls. In *Proceedings of IJCAI*, pages 3569–3576, 2016.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of TACAS*, pages 337–340, 2008.

[10] Wei Gao, Hengyi Lv, Qiang Zhang, and Dunbo Cai. Estimating the volume of the solution space of SMT(LIA) constraints by a flat histogram method. *Algorithms*, 11:142, 2018.

[11] Cunjing Ge, Feifei Ma, Tian Liu, Jian Zhang, and Xutong Ma. A new probabilistic algorithm for approximate model counting. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, 2018.

[12] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 54–61, 2006.

[13] Seonmo Kim and Stephen McCamant. Bit-vector model counting using statistical estimation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2018.

[14] Vladimir Klebanov, Alexander Weigl, and Jörg Weisbarth. Sound Probabilistic #SAT with Projection. In *Workshop on QAPL*, 2016.

[15] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proceedings of PLDI*, pages 565–576, 2014.

[16] Jean-Philippe Martin. Upper and lower bounds on the number of solutions. Technical Report MSR-TR-2007-164, Microsoft Research, 2007.

[17] Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of PLAS*, pages 1:1–1:12, 2011.

[18] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conf. on Artificial Intelligence*, 2012.

[19] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of SAT*, 2004.

[20] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019.

[21] Michael Sipser. A complexity theoretic approach to randomness. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, page 330–335, 1983.

[22] Mate Soos and Kuldeep Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[23] Larry Stockmeyer. The complexity of approximate counting. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, page 118–126, 1983.

[24] Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proceedings of SAT*, pages 424–429, 2006.

[25] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.