Fast Sampling of Perfectly Uniform Satisfying Assignments

Dimitris Achlioptas[1,2], Zayd S. Hammoudeh[1(✉)], and Panos Theodoropoulos[2]

[1] Department of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, USA
{dimitris,zayd}@ucsc.edu

[2] Department of Informatics and Telecommunications, University of Athens, Athens, Greece
ptheodor@di.uoa.gr

Abstract. We present an algorithm for perfectly uniform sampling of satisfying assignments, based on the exact model counter sharpSAT and reservoir sampling. In experiments across several hundred formulas, our sampler is faster than the state of the art by 10 to over 100,000 times.

1 Introduction

The DPLL [4] procedure forms the foundation of most modern SAT solvers. Its operation can be modeled as the preorder traversal of a rooted, binary tree where the root corresponds to the empty assignment and each edge represents setting some unset variable to 0 or 1, so that each node of the tree corresponds to a distinct partial assignment.

If the residual formula under a node's partial assignment is empty of clauses, or contains the empty clause, the node is a leaf of the tree. Naturally, the leaves corresponding to the former case form a partition of the formula's satisfying assignments (models), each part called a *cylinder* and having size equal to 2^z, where $z \geq 0$ is the number of unassigned variables at the leaf.

Generally, improved SAT solver efficiency is derived by trimming the DPLL search tree. For instance, conflict-driven clause learning (CDCL) amounts to adding new clauses to the formula each time a conflicting assignment is encountered. These added (learned) clauses make it possible to identify partial assignments with no satisfying extensions higher up in the tree.

1.1 Model Counting

Naturally, we can view model counting as the task where each internal node of the aforementioned tree simply adds the number of models of its two children. With this in mind, we see that the aforementioned CDCL optimization carries over, helping identify subtrees devoid of models sooner.

Research supported by NSF grants CCF-1514128, CCF-1733884, an Adobe research grant, and the Greek State Scholarships Foundation (IKY).

O. Beyersdorff and C. M. Wintersteiger (Eds.): SAT 2018, LNCS 10929, pp. 135–147, 2018.
https://doi.org/10.1007/978-3-319-94144-8_9

Despite the similarity with SAT solving, though, certain optimizations are uniquely important to efficient model counting. Specifically, it is very common for different partial assignments to have the same residual formula. While CDCL prevents the repeated analysis of unsatisfiable residual formulas, it does not prevent the reanalysis of previously encountered satisfiable residual formulas. To prevent such reanalysis #SAT solvers, e.g., Cachet [9], try to memoize in a cache the model counts of satisfiable residual formulas. Thus, whenever a node's residual formula is in the cache, the node becomes a leaf in the counting tree. We will refer to the tree whose leaves correspond to the execution of a model counter employing caching as a *compact counting* tree.

Another key optimization stems from the observation that as variables are assigned values, the formula tends to break up into pieces. More precisely, given a formula consider the graph having one vertex per clause and an edge for every pair of clauses that share at least one variable. Routinely, multiple connected components are present in the graph of the input formula. More importantly, as variables are assigned, components split. Trivially, a formula is satisfiable iff all its components are satisfiable. Determining the satisfiability of each component-formula independently can confer dramatic computational benefits [1].

The DPLL-based model counter sharpSAT [11], originally released in 2006 by Thurley and iteratively improved since, is the state-of-the-art exact model counter. It leverages all of the previously discussed optimizations and integrates advanced branch-variable selection heuristics proposed in [10]. Its main advantage over its predecessors stems from its ability to cache more components that are also of greater relevance. It achieves this through a compact encoding of cache entries as well as a cache replacement algorithm that takes into account the current "context", i.e., the recent partial assignments considered. Finally, it includes a novel algorithm for finding failed literals in the course of Boolean Constraint Propagation (BCP), called implicit BCP, which makes a very significant difference in the context of model counting.

Our work builds directly on top of sharpSAT and benefits from all the ideas that make it a fast exact model counter. Our contribution is to leverage this speed in the context of sampling. Generically, i.e., given a model counter as a black box, one can sample a satisfying assignment with $2n$ model counter invocations by repeating the following: pick an arbitrary unset variable v; count the number of models Z_0, Z_1, of the two formulas that result by setting v to 0,1, respectively; set v to 0 with probability $Z_0/(Z_0 + Z_1)$, otherwise set it to 1.

As we discuss in Sect. 4 it is not hard to improve upon the above by modifying sharpSAT so that, with essentially no overhead, it produces a *single* perfectly uniform sample in the course of its normal, model counting execution. Our main contribution lies in introducing a significantly more sophisticated modification, leveraging a technique known as reservoir sampling, so that with relatively little overhead, it can produce *many* samples. Roughly speaking, the end result is a sampler for which one can largely use the following rule of thumb:

Generating 1,000 perfectly uniform models takes
about 10 times as long as it takes to count the models.

2 Related Work

In digital functional verification design defects are uncovered by exposing the device to a set of test stimuli. These stimuli must satisfy several requirements to ensure adequate verification coverage. One such requirement is that test inputs be diverse, so as to increase the likelihood of finding bugs by testing different corners of the design.

Constrained random verification (CRV) [8] has emerged in recent years as an effective technique to achieve stimuli diversity by employing randomization. In CRV, a verification engineer encodes a set of design constraints as a Boolean formula with potentially hundreds of thousands of variables and clauses. A constraint solver then selects a random set of satisfying assignments from this formula. Efficiently generating these random models, also known as witness, remains a challenge [3].

Current state of the art witness generators, such as UniGen2 [2], use a hash function to partition the set of all witnesses into roughly equal sized groups. Selecting such a group uniformly at random and then a uniformly random element from within the selected group, produces an approximately uniform witness. This approximation of uniformity depends on the variance in the size of the groups in the initial hashing-based partition. In practical applications, this non-uniformity is not a major issue.

Arguably the main drawback of hash-based witness generators is that their total execution time grows *linearly* with the number of samples. Acceleration can be had via parallelization, but at the expense of sacrificing witness independence. Also, by their probabilistic nature, hash-based generators may fail to return the requested number of models.

Our tool SPUR (Satisfying Perfectly Uniformly Random) addresses the problem of generating many samples by combining the efficiencies of sharpSAT with reservoir sampling. This allows us to draw a very large number of samples per traversal of the compact counting tree.

3 Caching and Component Decomposition

Most modern #SAT model counters are DPLL-based, and their execution can be modeled recursively. For example, Algorithm 1 performs model counting with component decomposition and caching similar to sharpSAT [11]. (We have simplified this demonstrative implementation by stripping out efficiency enhancements not directly relevant to this discussion including CDCL, unit clause propagation, non-chronological backtracking, cache compaction, etc.)

The algorithm first looks for F and its count in the cache. If they are not there, then if F is unsatisfiable or empty, model counting is trivial. If F has multiple connected components, the algorithm is applied recursively to each one and the product of the model counts is returned. If F is a non-empty, connected formula not in the cache, then a branching variable is selected, the algorithm is applied to each of the two restricted subformulas, and the sum of the model counts is returned after it has been deposited in the cache along with F.

Algorithm 1. Model counting with component decomposition and caching

```
1: function CounterCounter(F)
2:     if IsCachedIsCached(F) then
3:         return CachedCountCachedCount(F)                          ▷ Cache-hit leaf
4:
5:     if UnsatUnsat(F) then return 0
6:     if ClausesClauses(F) = ∅ then return 2^|VarVar(F)|            ▷ Cylinder leaf
7:
8:     C_1, ..., C_k ← ComponentDecompositionComponentDecomposition(F)   ▷ Component decomposition
9:     if k > 1 then
10:        for i from 1 to k do
11:            Z_i ← CounterCounter(C_i)
12:        Z ← ∏_{i=1}^{k} Z_i
13:        return Z
14:
15:    v ← BranchVariableBranchVariable(F)
16:    Z_0 ← CounterCounter(F ∧ v = 0)
17:    Z_1 ← CounterCounter(F ∧ v = 1)
18:    Z ← Z_0 + Z_1
19:    AddToCacheAddToCache(F, Z)          ▷ The count of every satisfiable, connected
20:                                        ▷ subformula ever encountered is cached
21:    return Z
```

4 How to Get One Uniform Sample

It is easy to modify Algorithm 1 so that it returns a *single* uniformly random model of F. All we have to do is: (i) require the algorithm to return one model along with the count (ii) select a uniformly random model whenever we reach a cylinder, and (iii) at each branching node, when the two recursive calls return with two counts and two models, select one of the two models with probability proportional to its count, and store it along with the sum of the two counts in the cache before returning it. In the following, $F(\sigma)$ denotes the restriction of formula F by partial assignment σ and $\text{Free}(\sigma)$ denotes the variables not assigned a value by σ.

An important observation is that the algorithm does not actually need to select, cache, and return a random model every time it reaches a cylinder. It can instead simply return the partial assignment corresponding to the cylinder. After termination, we can trivially "fill out" the returned cylinder to a complete satisfying assignment. This can be a significant saving as, typically, there are many cylinders, but we only need to return one model. Algorithm 2 employs this idea so that it returns a cylinder (instead of a model), each cylinder having been selected with probability proportional to its size.

The correctness of Algorithm 2 would be entirely obvious in the absence of model caching. With it, for any subformula, F', we only select a model at most once. This is because after selecting a model τ of F' for the first time in line 20 we

write τ along with F' in the cache, in line 21, and therefore, if we ever encounter F' again, lines 2, 3 imply we will return τ as a model for F'. Naturally, even though we reuse the same model for a subformula encountered in completely different parts of the tree, no issue of probabilistic dependence arises: since we only return one sample overall, and thus for F', how could it?

It is crucial to note that this fortuitous non-interaction between caching and sampling does *not* hold for multiple samples, since if a subformula appears at several nodes of the counting tree, the sample models associated with these nodes must be independent of one another.

Algorithm 2. Single model sampler

```
1: function OneModel(F, σ)
2:     if IsCached(F(σ)) then
3:         return CachedCount(F(σ)), CachedModel(F(σ))
4:
5:     if Unsat(F(σ)) then return 0, –
6:     if Clauses(F(σ)) = ∅ then return 2^|Free(σ)|, σ
7:
8:     C_1, ..., C_k ← ComponentDecomposition(F(σ))
9:     if k > 1 then
10:        for i from 1 to k do
11:            Z_i, σ_i ← OneModel(C_i, σ)
12:        Z ← ∏_{i=1}^{k} Z_i
13:        τ ← σ_1, ..., σ_k
14:        return Z, τ
15:
16:    v ← BranchVariable(F(σ))
17:    Z^0, σ^0 ← OneModel(F, σ ∧ v = 0)
18:    Z^1, σ^1 ← OneModel(F, σ ∧ v = 1)
19:    Z ← Z^0 + Z^1
20:    τ ← σ^0 with probability Z^0/Z, otherwise τ ← σ^1
21:    AddToCache(F(σ), Z, τ)
22:
23:    return Z, τ
```

5 How to Get Many Uniform Samples at Once

Consider the set $\mathcal{C}$ which for each leaf σ_j of the compact counting tree comprises a pair (σ_j, c_j), where c_j is the number of satisfying extensions (models) of partial assignment σ_j. The total number of models, Z, therefore equals $\sum_j c_j$. Let $\mathrm{Bin}\,(n, p)$ denote the Binomial random variable with n trials of probability p.

To sample s models uniformly, independently, and with replacement (u.i.r.), we would like to proceed as follows:

1. Enumerate $\mathcal{C}$, while enjoying full model count caching, as in sharpSAT.
2. Without storing the (huge) set $\mathcal{C}$, produce from it a random set $\mathcal{R}$ comprising pairs $\{(\sigma_i, s_i)\}_{i=1}^{t}$, for some $1 \leq t \leq s$, such that:
   (a) Each σ_i is a distinct leaf of the compact counting tree.
   (b) $s_1 + \cdots + s_t = s$ (we will eventually generate s_i extensions of σ_i u.i.r.).
   (c) For every leaf σ_j of the compact counting tree and every $1 \leq w \leq s$, the probability that (σ_j, w) appears in $\mathcal{R}$ equals $\Pr[\text{Bin}\,(s, c_j/Z) = w]$.

Given a set $\mathcal{R}$ as above, we can readily sample models corresponding to those pairs (σ_i, s_i) in $\mathcal{R}$ for which either $s_i = 1$ (by invoking OneModel($F(\sigma_i)$)), or for which Clauses($F(\sigma_i)$) $= \emptyset$ (trivially). For each pair (σ_i, s_i) for which $s_i > 1$, we simply run the algorithm again on $F(\sigma_i)$, getting a set $\mathcal{R}'$, etc.

Obviously, the non-trivial part of the above plan is achieving (2c) without storing the (typically huge) set $\mathcal{C}$. We will do this by using a very elegant idea called reservoir sampling [12], which we describe next.

6 Reservoir Sampling

Let A be an arbitrary finite set and assume that we would like to select s elements from A u.i.r. for an arbitrary integer $s \geq 1$. Our task will be complicated by the fact that the (unknown) set A will not be available to us at once. Instead, let $A_1, A_2, \ldots, A_m$ be an arbitrary, unknown partition of A. Without any knowledge of the partition, or even of m, we will be presented with the parts in an arbitrary order. When each part is presented we can select some of its elements to store, but our storage capacity is precisely s, i.e., at any given moment we can only hold up to s elements of A. Can we build a sample as desired?

Reservoir sampling is an elegant solution to this problem that proceeds as follows. Imagine that (somehow) we have already selected s elements u.i.r. from a set B, comprising a multiset S. Given a set C disjoint from B we can produce a sample of s elements selected u.i.r. from $B \cup C$, without access to B, as follows. Note that in Step 3 of Algorithm 3, multiple instances of an element of B in S are considered distinct, i.e., removing one instance leaves the rest unaffected. It is not hard to see that after Step 4 the multiset S will comprise s elements selected u.i.r. from $B \cup C$. Thus, by induction, starting with $B = \emptyset$ and processing the sets $A_1, A_2, \ldots$ one by one (each in the role of C) achieves our goal.

Algorithm 3. Turns a u.i.r. s-sample $S \subseteq B$ to a u.i.r. s-sample of $B \cup C$

1: Generate $q \sim \text{Bin}\,(s, |C|/|B \cup C|)$.
2: Select q elements from C u.i.r.
3: Select q elements from S uniformly, independently, without replacement.
4: Swap the selected elements of S for the selected elements of C.

6.1 Reservoir Sampling in the Context of Model Caching

In our setting, each set A_i amounts to a leaf of the compact counting tree. We would like to build our sample set by (i) traversing this tree exactly as sharpSAT, and (ii) ensuring that every time the traversal moves upwards from a leaf, we hold s models selected u.i.r. from all satisfying extensions of leaves encountered so far. More precisely, instead of actual samples, we would like to hold a random set $\mathcal{R}$ of weighted partial assignments satisfying properties (2a)–(2c) in Sect. 5.

To that end, it will be helpful to introduce the following distribution. Given r bins containing $s_1, \ldots, s_r$ distinct balls, respectively, and $q \geq 0$ consider the experiment of selecting q balls from the bins uniformly, independently, without replacement. Let $\mathbf{q} = (q_1, \ldots, q_r)$ be the (random) number of balls selected from each bin. We will write $\mathbf{q} \sim \mathcal{D}((s_1, \ldots, s_r), q)$. To generate a sample from this distribution, let $b_0 = 0$; for $i \in [r]$, let $b_i = s_1 + \cdots + s_i$, so that $b_1 = s_1$ and $b_r = s_1 + \ldots + s_r := s$. Let $\gamma_1, \gamma_2, \ldots, \gamma_q$ be i.i.d. uniform elements of $[s]$. Initialize q_i to 0 for each $i \in [r]$. For each $i \in [q]$: if $\gamma_i \in (b_{z-1}, b_z]$, then increment q_r by 1.

With this in mind, imagine that we have already processed t leaves so that $Z_t = Z = |A_1| + \cdots |A_t|$ and that the reservoir contains $\mathcal{R} = \{(\sigma_i, s_i)\}_{i=1}^r$, such that $\sum_{i=1}^r s_i = s$. Let σ be the current leaf (partial assignment), let A be the set of σ's satisfying extensions, and let $w = |A|$. To update the reservoir, we first determine the random number, $q \geq 0$, of elements from A to place in our s-sample, as a function of w, Z. Having determined q we draw from $\mathcal{D}((s_1, \ldots, s_r), q)$ to determine how many elements to remove from each set already in the reservoir, by decrementing its weight s_i (if $s_i \leftarrow 0$ we remove $(\sigma_i, 0)$ from the reservoir). Finally, we add (σ, q) to the reservoir to represent the q elements of A.

Note that, in principle, we could have first selected s elements u.i.r. from A and then $0 \leq q \leq s$ among them to merge into the reservoir (again represented as (σ, q)). This viewpoint is useful since, in general, instead of merging into the existing reservoir $0 \leq q \leq s$ elements from a single cylinder of size w, we will need to merge q elements from a set of size w that is the union of $\ell \geq 1$ disjoint sets, each represented by a partial assignment σ_j, such that we have already selected a_j elements from each set, where $\sum_{j=1}^{\ell} a_i = s$. Indeed, Algorithm 4 below is written with this level of generality in mind, so that our simple single cylinder example above corresponds to merging $\langle w, \{(\sigma, s)\}\rangle$ into the reservoir.

Algorithm 4. Merges $R = \langle Z, \{(\sigma_i, s_i)\}_{i=1}^r \rangle$ with $\langle w, \{(\sigma_j, a_j)\}_{j=1}^{\ell} \rangle$

1: **function** ReservoirUpdate($R, \langle w, \{(\sigma_j, a_j)\}_{j=1}^{\ell} \rangle$)
2: $Z \leftarrow Z + w$
3: $q \sim \mathrm{Bin}(s, w/Z)$
4: Generate $(\beta_1, \ldots, \beta_\ell) \sim \mathcal{D}((a_1, \ldots, a_\ell), q)$
5: Generate $(\gamma_1, \ldots, \gamma_r) \sim \mathcal{D}((s_1, \ldots, s_r), q)$
6: $R' \leftarrow \langle Z, \{(\sigma_j, \beta_j)\}_{j=1}^{\ell} \cup \{(\sigma_i, s_i - \gamma_i)\}_{i=1}^r \rangle$
7: Discard any partial assignment in R' whose weight is 0
8: **return** R'

7 A Complete Algorithm

To sample s models u.i.r. from a formula F, we create an empty reservoir R of capacity s and invoke SPUR$(F, \emptyset, R)$. The call returns the model count of F and modifies R in place to contain pairs $\{(\sigma_i, s_i)\}_{i=1}^{t}$, for some $1 \leq t \leq s$, such that $\sum_{i=1}^{t} s_i = s$. Thus, SPUR partitions the task of generating s samples into t independent, smaller sampling tasks. Specifically, for each $1 \leq i \leq t$, if Clauses$(F(\sigma_i)) = \emptyset$, then sampling the s_i models is trivial, while if $s_i = 1$, sampling can be readily achieved by invoking OneModel on $F(\sigma_i)$. If none of the two simple cases occurs, SPUR is called on $F(\sigma_i)$ requesting s_i samples.

Algorithm 5. Counts models and fills up a reservoir with s samples

```
1: function SPUR(F, σ, R)
2:     if IsCached(F(σ)) then
3:         ReservoirUpdate(R, ⟨CachedCount(F(σ)), (σ, s)⟩)
4:         return CachedCount(F(σ))
5:
6:     if Unsat(F(σ)) then return 0
7:     if Clauses(F(σ)) = ∅ then
8:         ReservoirUpdate(R, ⟨2^|Free(σ)|, (σ, s)⟩)
9:         return 2^|Free(σ)|
10:
11:    C_1, ..., C_k ← ComponentDecomposition(F(σ))
12:    if k > 1 then
13:        for i from 1 to k do
14:            Create a new reservoir R_i of capacity s
15:            Z_i ← SPUR(C_i, ∅, R_i)
16:        w ← ∏_{i=1}^{k} Z_i
17:        A ← Stitch(σ, R_1, R_2, ..., R_k)
18:        ReservoirUpdate(R, ⟨w, A⟩)
19:        return w
20:
21:    v ← BranchVariable(F(σ))
22:    Z_0 ← SPUR(F, σ ∧ v = 0, R)
23:    Z_1 ← SPUR(F, σ ∧ v = 1, R)
24:    AddToCache(F, Z_0 + Z_1)
25:    return Z_0 + Z_1
```

If a formula has $k > 1$ components, SPUR is invoked recursively on each component C_i with a new reservoir R_i (also passed by reference). When the recursive calls return, each reservoir R_i comprises some number of partial assignments over the variables in C_i, each with an associated weight (number of samples), so that the sum of the weights equals s. It will be convenient to think of the content of each reservoir R_i as a multiset containing exactly s strings from $\{0, 1, *\}^{\mathrm{Var}(C_i)}$. Under this view, to Stitch together two reservoirs R_1, R_2, we fix an arbitrary

permutation of the s strings in, say, R_1, pick a uniformly random permutation of the strings in R_2, and concatenate the first string in R_1 with the first string in R_2, the second string in R_1 with the second string in R_2, etc. To stitch together multiple reservoirs we proceed associatively. The final result is a set $\{(\sigma_j, a_j)\}_{j=1}^{\ell}$, for some $1 \leq \ell \leq s$, such that $\sum_{j=1}^{\ell} a_j = s$.

8 Evaluation and Experiments

We have developed a prototype C++ implementation [6] of SPUR on top of sharpSAT (ver. 5/2/2014) [11]. This necessitated developing multiple new modules as well as extensively modifying several of the original ones.

8.1 Uniformity Verification

Since sharpSAT is an exact model counter, the samples derived from SPUR are perfectly uniform. Since we use reservoir sampling, they are also perfectly independent. As a test of our implementation we selected 55 formulas with model counts ranging from 2 to 97,536 and generated 4 million models of each one.

For each formula F we (i) recorded the number of times each of its $M(F)$ models was selected by SPUR, and (ii) drew 4 million times from the multinomial distribution with $M(F)$ outcomes, corresponding to ideal u.i.r. sampling. We measured the KL-divergence of these two empirical distributions from the multinomial distribution with $M(F)$ outcomes, so that the divergence of the latter provides a yardstick for the former. The ratio of the two distances was close to 1 over all formulas, and the product of the 55 ratios was 0.357.

One of the formulas we considered was `case110` with 16,384 models, which was used in the verification of the approximate uniformity of UniGen2 in [2]. Figure 1 plots the output of UniGen2 and SPUR against a background of the ideal multinomial distribution (with mean 244.14...). Each point (x, y) represents the number of models, x, that were generated y times across all 4,000,000 trials.

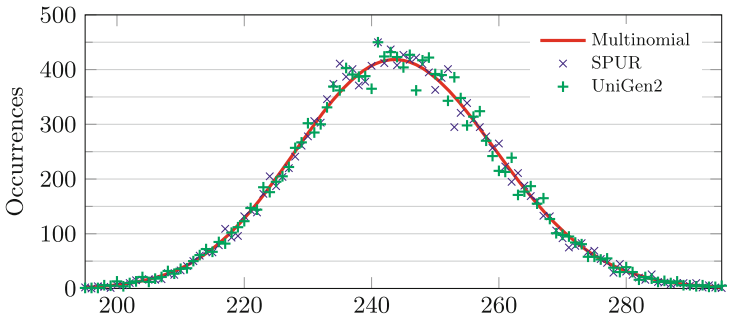

Fig. 1. Uniformity comparison between an ideal uniform sampler, SPUR and UniGen2 on the "`case110`" benchmark on four million samples.

8.2 Running Time

To demonstrate the empirical performance of SPUR we ran it on several hundred formulas, along with UniGen2 (ver. 9/28/2017), an almost-uniform, almost-i.i.d. SAT witness generator, representing the state of the art prior to our work.

Benchmarks: We considered 586 formulas, varying in size from 14 to over 375,000 variables. They are the union of the 369 formulas used to benchmark UniGen2 in [7] (except for approximately 20 proprietary formulas with suffix `_new` that are not publicly available) and the 217 formulas used to benchmark sharpSAT in [11]. Of the latter we removed from consideration the 100 formulas in the `flat200` graph coloring dataset, since on all of them UniGen2 timed out, while SPUR terminated successfully in a handful of seconds. This left 486 formulas.

An important distinction between the two sets of formulas is that all formulas from [7] come with a *sampling set*, i.e., a relatively small subset, S, of variables. When such a set is given as part of the input, UniGen2 samples (near-)uniformly from the elements of $\{0,1\}^S$ that have at least one satisfying extension (model). For all but 17 of the 369 formulas, the provided set was in fact an *independent support* set, i.e., each of element of $\{0,1\}^S$ was guaranteed to have at most one satisfying extension. Thus for these 352 formulas UniGen2 is, in fact, sampling satisfying assignments, making them fair game for comparison (if anything such formulas slightly favor UniGen2 as we do not include the time required to extend the returned partial assignments to full assignments which, in principle, could be substantial.) None of the 117 formulas used to benchmark sharpSAT come with such a set (since sharpSAT does not support counting the size of projections of the set of models). Of these $486 - 17 = 469$ formulas, 2 are unsatisfiable, while for another 22 UniGen2 crashed or exited with an error. (SPUR did not crash or report an error on any formulas.) Of the remaining 445 formulas, 72 caused both SPUR and UniGen2 to time out. We report on the remaining 373 formulas.

For each formula we generated between 1,000 and 10,000 samples, as originally performed by Chakraborty et al. [2] and report the results in detail. Our main finding is that SPUR is on average more than 400× faster than UniGen2, i.e., the geometric mean[1] of the speedup exceeds 400×. We also compared the two algorithms when they only generate 55 samples per formula. In that setting, the geometric mean of the speedup exceeds 150×.

Experiment Setup: All experiments were performed on a high-performance cluster, where each node consists of two Intel Xeon E5-2650v4 CPUs with up to 10 usable cores and 128 GB of DDR4 DRAM. All our results were generated on the same hardware to ensure a fair comparison. UniGen2's timeout was set to 10 h; all other UniGen2 hyperparameters, e.g., κ, `startIteration`, etc., were left at their default values. The timeout of SPUR was set to 7 h and its maximum cache size was set to 8 GB. All instances of the two programs run on a single core at a time.

[1] The arithmetic mean [of the speedup] is even greater (always). For the aptness of using the geometric mean to report speedup factors see [5].

8.3 Comparison

Table 1 reports the time taken by SPUR and UniGen2 to generate 1,000 samples for a representative subset of the benchmarks. Included in the table is also the speedup factor of SPUR relative to UniGen2, i.e., the ratio of the two execution times. Since sharpSAT represents the execution time floor for SPUR we also provide the ratio between SPUR's execution time and of a *single* execution of sharpSAT. Numbers close to 1 substantiate the heuristic claim "if you can count the models with sharpSAT, you can sample."

Table 1. Time (sec) comparison of SPUR and UniGen2 to generate 1,000 samples.

Benchmark	#Var	#Clause	$\frac{\text{SPUR}}{\text{sharpSAT}}$	UniGen2 (sec)	SPUR (sec)	Speedup
case5	176	518	19.1	633	0.84	753
registerlesSwap	372	1,493	7.0	28,778	0.26	110,684
s953a_3_2	515	1,297	13.4	1,139	1.03	1,105
s1238a_3_2	686	1,850	7.0	610	2.31	264
s1196a_3_2	690	1,805	10.0	516	2.10	245
s832a_15_7	693	2,017	13.5	56	0.81	69
case_1_b12_2	827	2,725	1.4	689	29	23
squaring30	1,031	3,693	3.7	1,079	4.58	235
27	1,509	2,707	1.0	99	0.017	5,823
squaring16	1,627	5,835	1.9	11,053	78	141
squaring7	1,628	5,837	1.4	2,185	38	57
111	2,348	5,479	1.0	163	0.029	5,620
51	3,708	14,594	1.5	714	0.11	6,490
32	3,834	13,594	1.0	181	0.051	3,549
70	4,670	15,864	1.0	196	0.056	3,500
7	6,683	24,816	1.0	173	0.077	2,246
Pollard	7,815	41,258	6.0	181	355	0.51
17	10,090	27,056	1.6	192	0.092	2,086
20	15,475	60,994	2.7	289	2.05	140
reverse	75,641	380,869	6.2	TIMEOUT	2.66	>13,533

Figure 2 compares the time required to generate 1,000 witnesses with SPUR and UniGen2 for the full set of 373 benchmarks. The axes are logarithmic and each mark represents a single formula. Formulas for which a timeout occurred appear along the top or right border, depending on which tool timed out. (For marks corresponding to timeouts, the axis of the tool for which there was a timeout was co-opted to create a histogram of the number of timeouts that occurred.) These complete results can be summarized as follows:

- SPUR was faster than UniGen2 on 371 of the 373 benchmarks.
- On 369 of the 373, SPUR was more than 10× faster.

- On over 2/3 of the benchmarks, it was more than 100× faster.
- The geometric mean of the speedup exceeds 400×.
- On over 70% of the benchmarks, SPUR generated 1,000 samples within at most 10× of a single execution of sharpSAT.
- SPUR was 3 times more likely than UniGen2 to successfully generate witnesses for large formulas, (e.g., >10,000 variables).

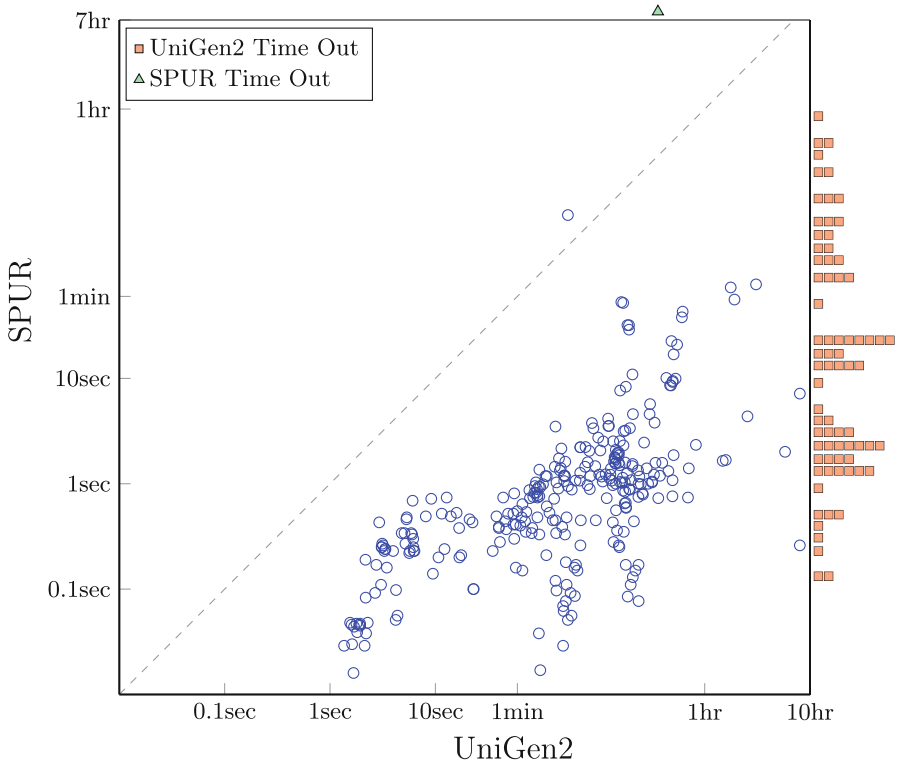

Fig. 2. Comparison of the running time to generate 1,000 samples between UniGen2 and SPUR over 373 formulas.

References

1. Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, pp. 157–162. AAAI Press (2000)
2. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 304–319. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_25

3. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proceedings of the 51st Annual Design Automation Conference, DAC 2014, pp. 60:1–60:6. ACM, New York (2014). http://doi.acm.org/10.1145/2593069.2593097
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
5. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM **29**(3), 218–221 (1986). http://doi.acm.org.oca.ucsc.edu/10.1145/5666.5673
6. SPUR source code. https://github.com/ZaydH/spur
7. Meel, K.: Index of UniGen verification benchmarks. https://www.cs.rice.edu/CS/Verification/Projects/UniGen/Benchmarks/
8. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. In: Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence, IAAI 2006, vol. 2, pp. 1720–1727. AAAI Press (2006)
9. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004 (2004)
10. Sang, T., Beame, P., Kautz, H.: Heuristics for fast exact model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 226–240. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_17
11. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_38
12. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985). http://doi.acm.org/10.1145/3147.3165