Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling

Mate Soos
National University of Singapore
Singapore

Kuldeep S. Meel
National University of Singapore
Singapore

ABSTRACT

Given a Boolean formula φ over the set of variables X and a projection set $\mathcal{P} \subseteq X$, then if $\mathcal{I} \subseteq \mathcal{P}$ is independent support of $\mathcal{P}$, then if two solutions of φ agree on $\mathcal{I}$, then they also agree on $\mathcal{P}$. The notion of independent support is related to the classical notion of definability dating back to 1901, and have been studied over the decades. Recently, the computational problem of determining independent support for a given formula has attained importance owing to the crucial importance of independent support for hashing-based counting and sampling techniques.

In this paper, we design an efficient and scalable independent support computation technique that can handle formulas arising from real-world benchmarks. Our algorithmic framework, called Arjun[1], employs implicit and explicit definability notions, and is based on a tight integration of gate-identification techniques and assumption-based framework. We demonstrate that augmenting the state-of-the-art model counter ApproxMC4 and sampler UniGen3 with Arjun leads to significant performance improvements. In particular, ApproxMC4 augmented with Arjun counts 576 more benchmarks out of 1896 while UniGen3 augmented with Arjun samples 335 more benchmarks within the same time limit.

ACM Reference Format:
Mate Soos and Kuldeep S. Meel. 2022. Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549406

1 INTRODUCTION

Given a Boolean formula φ over the set of variables X, let $sol(\varphi)$ represent the set of solutions of φ. For a given assignment σ over X and a subset of variables $\mathcal{P} \subseteq X$, let $\sigma_{\downarrow \mathcal{P}}$ represent the assignment of variables restricted to $\mathcal{P}$. Given a Boolean formula φ over the set of variables X and a projection set $\mathcal{P} \subseteq X$, a subset of variables $\mathcal{I}$ such that $\mathcal{I} \subseteq \mathcal{P}$ is called independent support of $\mathcal{P}$ if $\forall \sigma_1, \sigma_2 \in sol(\varphi), \sigma_{1\downarrow \mathcal{I}} = \sigma_{2\downarrow \mathcal{I}} \implies \sigma_{1\downarrow \mathcal{P}} = \sigma_{2\downarrow \mathcal{P}}$. In this paper, we focus on the design of efficient algorithmic techniques to compute $\mathcal{I}$ for a given φ and $\mathcal{P}$.

[1]The tool is available open-source at https://github.com/meelgroup/arjun

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/3508352.3549406

1.1 Applications: Counting and Sampling

Given a Boolean formula φ and a projection set $\mathcal{P} \subseteq X$, the problem of projected model counting seeks to compute the number of solutions of the formula $\exists Y \varphi$ where $Y = X \setminus \mathcal{P}$; alternatively, the solutions of the formula $\exists Y \varphi$ can be viewed as solutions of the formula projected on $\mathcal{P}$. Observe that projected model counting is a strict generalization of the classical problem of propositional model counting which focuses on the case when $\mathcal{P} = X$. Similarly, the problem of projected sampling seeks to uniformly sample solutions of $\exists Y \varphi$.

Projected counting and sampling are fundamental problems in computer science with a wide variety of applications ranging from network reliability [9], neural network verification [1], computational biology [25], and software and hardware testing [21]. For example, given a neural network $\mathcal{N}$ and a property ψ, the problem of estimating how often the network satisfies the property ψ reduces to projected counting [1].

From a theoretical viewpoint, projected counting is #NP-complete; it is worth remarking that the problem of propositional model counting is #P-complete [30], and it is known that #P $\subseteq$ #NP. The hardness of #P (and #NP) motivated efforts towards approximation methods with (ε, δ)-guarantees. The state of the art approximate techniques for counting and sampling are hashing-based [5–7, 28, 29], seeking to combine the power of universal hashing with SAT solving.

The core idea of hashing-based counting techniques is to employ pairwise independent hash functions, also known as strongly universal hash functions, to partition the solution space into *roughly equal small* cells of solutions and then pick a cell randomly. The number of solutions in a cell can then be determined exactly by enumerating the solutions in the cell one by one, in case the cell is *small*. The number of solutions of the formula is then estimated as the number of solutions of a *randomly chosen small* cell multiplied by the number of cells. In case of sampling, one enumerates the solutions in the cell and chooses one of the solutions at random.

To achieve estimates with rigorous (ε, δ)-guarantees (formally defined in Section 2), we rely on XOR-based 3-wise independent hash functions. A randomly picked hash function $h : \{0,1\}^n \mapsto \{0,1\}^m$ is explicitly represented as a conjunction of m randomly chosen XOR constraints, where each constraint is constructed by picking every variable with probability $\frac{1}{2}$. Therefore, the expected size of each XOR is $\frac{n}{2}$ where $n = |\mathcal{P}|$. Consequently, the SAT solver is invoked to find solutions of a formula expressed as conjunction of the original formula, φ, and the XOR constraints.

Runtime performance of hashing-based counting and sampling techniques is primarily determined by the time taken by the SAT solver as over 99% of the time is spent inside SAT calls. Accordingly, the past decade has witnessed a sustained effort in designing *sparse* XOR-based hash functions [7, 11, 12, 19]. An important advance

in this direction was achieved by Chakraborty et al. [7], who proposed the notion of independent support and observed that one can construct XORs only over the given independent support. It is worth remarking that Chakraborty et al. had defined the notion of independent support $\mathcal{P} = X$ but one can see that the notion easily generalizes to arbitrary $\mathcal{P}$.

1.2 Techniques to Identify Independent Support

Ivrii et al. [14] showed that the problem of computation of Minimal Independent Support can be reduced to Group Minimal Unsatisfiable Subset (GMUS), and the corresponding tool, MIS, was shown to scale to *moderately complex* instances. Concurrently, Lagniez et al. [15, 16] published a pre-processing technique, B+E, that combines the assumption-based framework offered by modern CDCL solvers with Padoa's theorem [23] to identify an independent support. An important conceptual viewpoint put forth by Lagniez et al. was the argument of eschewing the search for minimal independent support and instead focus on efficiently finding an independent support that may not be necessarily minimal.

1.3 Our Contributions

Our primary contribution is the design of an efficient framework, called Arjun, to identify independent support. Arjun consists of two phases. The first phase employs a syntactic gate recovery-based strategy, while the second phase is based on the tight integration of the standard assumption-based CDCL framework with the invocations needed to execute on Padoa's Theorem [23].

To demonstrate the efficacy of our approach, we perform a detailed empirical analysis over an extensive set of 1896 benchmarks arising from a diverse set of domains and employed in the analysis of counting and sampling techniques. We first showcase that with a timeout of 5000 seconds, Arjun can compute independent support for 358 more instances than the state of the art tool, MIS.

We observe that ApproxMC4 augmented with Arjun can count 1799 benchmarks, achieving an improvement of 576 benchmarks over ApproxMC4. Similarly, we observe that UniGen3 augmented with Arjun can sample 1372 benchmarks, achieving an improvement of 335 benchmarks within the same time limit of 5000s. Furthermore, we observe a significant improvement in runtime performance as well, in particular, the PAR-2 scores[2][2] for ApproxMC4 and UniGen3 augmented with Arjun are 575 and 2861 while the PAR-2 scores for ApproxMC4 and UniGen3 are 3717 and 4678, respectively.

Organization. The rest of the paper is organized as follows: We provide a detailed background in Section 2 and then introduce related work in Section 3. We then present a detailed algorithmic description of Arjun in Section 4. We present an extensive empirical evaluation in Section 5 and finally conclude in Section 6.

2 BACKGROUND

Given a directed graph $G = (V, E)$, a feedback vertex set $S \subseteq V$ is the set of vertices whose removal makes the graph acyclic. For an edge $e = (u, v)$, we say the edge e is outgoing from u and incoming to v. For a directed graph G, we use $\text{Root}(G)$ to denote the set of vertices in G which do not have any incoming edge.

Let $X = \{x_1, x_2 \dots x_n\}$ be the set of Boolean variables. Let a *literal* be a Boolean variable or its negation. A formula, φ, defined over X is known as a Conjunctive Normal Form (CNF), if φ is a conjunction of *clauses*, where each *clause* is disjunction of literals.

A *satisfying assignment* σ of φ is a mapping of $X \to \{0, 1\}$, such that φ evaluates True at σ. We often represent σ as the set of literals. We use $\sigma \models \varphi$ to denote σ as a *solution* or a *satisfying assignment* of φ. Furthermore, for $\mathcal{P} \subseteq X$, $\sigma_{\downarrow \mathcal{P}}$ represents the assignment of variables restricted to $\mathcal{P}$. We denote the set of all witnesses of φ by $sol(\varphi)$, and use $sol(\varphi)_{\downarrow \mathcal{P}}$ to indicate the projection of $sol(\varphi)$ on $\mathcal{P}$. We denote the subformula of φ that only contains clauses with the literal x by φ_x. We use the standard connectives $\vee$ for boolean OR, and $\wedge$ for boolean AND functions. We use the notation $\varphi|x$ to denote the formula φ when $x = \top$.

Example: consider $X = \{x_1, x_2, x_3\}$ and let $\sigma = (x_1, \neg x_2, x_3)$ (implying that x_1 and x_3 map to True while x_2 maps to False). Let $\mathcal{P} = \{x_1, x_2\}$, then $\sigma_{\downarrow P} = \{x_1, \neg x_2\}$.

Example 2: consider $\varphi = \{x_1 \vee x_3, \neg x_1 \vee x_3 \vee x_5, x_4 \vee x_5\}$, then $\varphi_{x_3}|x_1 = \{x_3 \vee x_5\}$.

2.1 Definability and Independent Support

Definition 1. *A subset of variables $\mathcal{I} \subseteq \mathcal{P}$ is an independent support of $\mathcal{P}$; if $\forall \sigma_1, \sigma_2 \in sol(\varphi)$, we have $\sigma_{1_{\downarrow \mathcal{I}}} = \sigma_{2_{\downarrow \mathcal{I}}} \implies \sigma_{1_{\downarrow \mathcal{P}}} = \sigma_{2_{\downarrow \mathcal{P}}}$*

Furthermore, $\mathcal{I} \subseteq \mathcal{P}$ is called minimal independent support of $\mathcal{P}$ if there does not exist $\hat{\mathcal{I}} \subset \mathcal{I}$ such that $\hat{\mathcal{I}}$ is an independent support.

Observe that if $\mathcal{I}$ is an independent support of $\mathcal{P}$, then we have $|sol(\varphi)_{\downarrow \mathcal{I}}| = |sol(\varphi)_{\downarrow \mathcal{P}}|$. The notion of independent support is related to the classical notion of definability. To this end, we first present the following two equivalent notions of definability.

Definition 2 (Implicit Definability). *A variable $x \in \mathcal{P}$ is implicitly defined by $\mathcal{I}$ for the formula φ if and only if $\forall \tau \in 2^{\mathcal{I}}$, we have $\varphi \wedge \tau \models x$ or $\varphi \wedge \tau \models \neg x$.*

Definition 3 (Explicity Definability). *A variable $x \in \mathcal{P}$ is explicitly defined by $\mathcal{I}$ for the formula φ if and only if there exists $\phi(\mathcal{I})$ such that $\varphi \models x \leftrightarrow \phi(\mathcal{I})$*

Lemma 2.1 (Beth's Theorem [3]). *A variable $x \in \mathcal{P}$ is explicitly defined by $\mathcal{I}$ for the formula φ if and only if x is implicitly defined by $\mathcal{I}$ for the formula φ.*

Since implicit and explicit definability are equivalent, we can omit referencing them and simply make statements such as: x is defined by $\mathcal{I}$ for the formula φ. Furthermore, the following remark follows from the notion of Independent support.

Remark 1. *If $\mathcal{I}$ is an independent support of $\mathcal{P}$ then all the variables $\mathcal{P} \setminus \mathcal{I}$ are defined by $\mathcal{I}$.*

2.2 Counting and Sampling

The problem of *propositional model counting* is to compute $|sol(\varphi)|$ for a given CNF formula φ. A *probably approximately correct* (or

[2]PAR-2 scores are used in the SAT competitions to measure performance. Each benchmark contributes a score that is the number of seconds used to finish (e.g. solve, count, sample) it, or in case of a timeout or memory out, twice the timeout in seconds. We then calculate the average score for all benchmarks, giving PAR-2.

PAC) counter is a probabilistic algorithm ApproxCount$(\cdot,\cdot,\cdot)$ that takes as inputs a formula φ, a tolerance $\varepsilon > 0$, and a confidence $1-\delta \in (0,1]$, and returns a count c with (ε,δ)-guarantees, i.e., $\Pr\left[\frac{|sol(\varphi)|}{(1+\varepsilon)} \leq c \leq (1+\varepsilon)|sol(\varphi)|\right] \geq 1-\delta$. Projected model counting is defined analogously using $sol(\varphi)_{\downarrow\mathcal{P}}$ instead of $sol(\varphi)$, for a given projection set[3] $\mathcal{P} \subseteq X$.

A *uniform sampler* outputs a solution $y \in sol(\varphi)$ such that $\Pr[y \text{ is output}] = \frac{1}{|sol(\varphi)|}$. An *almost-uniform sampler* relaxes the guarantee of uniformity and in particular, ensures that $\frac{1}{(1+\varepsilon)|sol(\varphi)|} \leq \Pr[y \text{ is output}] \leq \frac{1+\varepsilon}{|sol(\varphi)|}$.

2.3 The solve Algorithm of MiniSat

The second phase of Arjun focuses on the algorithmic engineering in the assumption-based CDCL framework. Therefore, to provide context to our contribution, we provide a brief overview of the standard assumption-based CDCL framework. To this end, the pseudocode sketch for solve(assumps) function is shown in Algorithm 1, presented in the style of the seminal MiniSat paper [10].

The solve(assumps) algorithm makes use of the following standard data structures and subroutines:

branch_depth current decision level, incremented every time the subroutine new_decision_level() is called.

backtrack_until$(\cdot)$ subroutine to perform the backtracking to the level given as the argument.

pick_branch_literal$(\cdot)$ picks an unassigned variable and corresponding value to be assigned. Typically, state of the art solvers use variants of VSIDS [18] to pick a branch variable and polarity caching [24] to decide the polarity of the literal.

value$(\cdot)$ value of the literal assigned currently. If a variable x is assigned True, then value(x) returns True while value($\neg$x) returns False.

analyze_conflict$(\cdot)$ Performs the conflict analysis and adds the learnt clause

should_restart() returns True or False based on the underlying restart policy

We now provide a quick overview of solve(assumps). First observe that there is an outer while loop 1– 27 and the algorithm exits either when a satisfying assignment is found or it can conclude that the formula is UNSAT. As noted earlier, we run CDCL loops up to a fixed number of conflicts. From our perspective, we focus on the lines 3– 10 wherein the algorithm inserts the assumptions into its decision queue as if it was a decision. Note that subsequent to each insertion of an assumption, a full, until-fixedpoint propagate() is called, on line 18.

3 RELATED WORK

Padoa's theorem provides a necessary and sufficient condition to determine whether x is defined by $\mathcal{I}$ for the formula φ. Let $\varphi(X)$ be defined on $X = \{x_1, x_2, \ldots x_n\}$, and $\mathcal{P}$ be of size t. Without loss of generality, let $\mathcal{P} = \{x_1, x_2, \ldots x_t\}$. We create another set of *fresh* variables $\hat{\mathcal{P}} = \{y_1, y_2, \ldots y_t\}$. Let $\varphi(\mathcal{P} \mapsto \hat{\mathcal{P}})$ represent the formula where every $x_i \in \mathcal{P}$ in φ is replaced by $y_i \in \hat{\mathcal{P}}$.

[3]Projection set has been referred to as sampling set in prior work [7, 20]

Algorithm 1 solve(assumps)

```
1: while True do
2:     branch ← None
3:     for i ← branch_depth; i < assumps.size() and branch is None; i++ do
4:         lit ← assumps[i]
5:         if value(lit) is False then
6:             backtrack_until(0); return UNSAT
7:         if value(lit) is True then
8:             new_decision_level()                    ▷ Fake decision level
9:             continue
10:        branch ← lit
11:    if branch is None then
12:        branch ← pick_branch_literal()
13:        if branch is None then                      ▷ Solution found
14:            save_assignment();backtrack_until(0); return SAT
15:    new_decision_level()
16:    enqueue(branch)
17: prop:
18:    ret ← propagate()
19:    if ret = conflict then
20:        analyze_conflict()
21:        if found_empty_conflict() then
22:            backtrack_until(0); return UNSAT
23:        backtrack_until(compute_backjump_level())
24:        go to prop
25:    if should_restart() then backtrack_until(0)
26:    if conflict_limit() then
27:        backtrack_until(0); return UNKNOWN
```

Lemma 3.1 (Padoa's Theorem [23]).

$$\psi(X,\hat{\mathcal{P}},i) := \varphi(X) \wedge \varphi(\mathcal{P} \mapsto \hat{\mathcal{P}}) \wedge \bigwedge_{\substack{j=1 \\ j\neq i}}^{t} (x_j \leftrightarrow y_j) \wedge x_i \wedge \neg y_i$$

A variable $x_i \in \mathcal{P}$ is defined by $\mathcal{I}$ for the formula φ iff $\psi(X,\hat{\mathcal{P}},i)$ is unsatisfiable.

Remark 2. *For a given formula φ, if variable $x_i \in \mathcal{P}$ is defined by $\mathcal{I}$ and a variable $x_j \in \mathcal{I}$ is defined by $\mathcal{I} \setminus x_j$, then x_i is defined by $\mathcal{I} \setminus x_j$.*

Combining the above observation with Padoa's theorem, Lagniez et al. [15] proposed an iterative procedure that performs $|\mathcal{P}|$ calls to a SAT oracle to determine a minimal independent support. To improve the efficiency, they propose to invoke a SAT solver with a set number of conflicts and to treat SAT or timeout as equivalent. Note that the presence of a pre-defined limit on conflicts causes the loss of guarantee of minimality for the independent support that is returned by the technique. However, such a loss of minimality is at the gain of efficiency in identifying a *small enough* independent support. Based on their paper, [15] published the tool B + E that performs the extraction of minimal independent support given an input CNF.

In another line of work, Ivrii et al. [14] reduced the problem of minimal independent support to Group Minimal Unsatisfiable Subset (GMUS) [17]. While the past decade has witnessed significant advances in the development of efficient GMUS tools, scalability remains a challenge. As a result, while Ivrii et al's proposed tool,

MIS, works exceedingly well for easy- to moderate-complexity formulas, it has difficulties with harder CNF formulas.

Recently, Slivovsky [27] observed that the resolution proof for unsatisfiability of $\psi(X, \hat{\mathcal{P}}, i)$ can be employed to generate the definition ϕ_i and x_i such that $\varphi \models x_i \leftrightarrow \phi_i(\mathcal{I})$. Furthermore, Slivovsky used all such extracted ϕ_i-s to perform pre-processing in the context of QBF.

4 ELEMENTS OF A FAST INDEPENDENT SUPPORT CALCULATOR

In this section, we delve into the primary technical contribution of this paper: Arjun, an efficient technique to compute independent support. Arjun consists of two phases. Each phase takes the formula φ and a projection set $\mathcal{P}$ as input and returns a set $\mathcal{I}$ such that $\mathcal{I} \subseteq \mathcal{P}$ is an independent support of $\mathcal{P}$. The two phases can be composed sequentially, by feeding the output of one phase as an input projection set to another phase.

The first phase employs a gate identification-based strategy, while the second phase is based on a tight integration of assumption-based framework to efficiently perform invocations based on Padoa's Theorem. It is worth emphasizing that while Beth's theorem asserts that both implicit and explicit definability are equivalent notions, our framework seeks to exploit the observation that there exist two classes of definable (or dependent) variables: one for which it is easy to extract their explicit definitions while for the other class of variables, we rely on the check for implicit definability via Padoa's Theorem.

4.1 Explicit Definability-based Identification

Given a formula φ and a projection set $\mathcal{P}$, we focus on finding the subset of variables $\mathcal{I} \subseteq \mathcal{P}$ along with the set of definitions Φ such that for every $x \in \mathcal{I}$, there is a corresponding definition $\phi \in \Phi$ such that ϕ is defined over $\mathcal{I}$.[4] For example, the set of clauses $\{\neg a \vee b, \neg a \vee c, a \vee \neg b \vee \neg c\}$ define variable a as $a = b \wedge c$ (i.e. an AND gate). Hence, if $a, b, c \in \mathcal{P}$ then we could set $\mathcal{I}$ to be $\mathcal{P} \setminus a$.

We observe that syntax-based gate identification techniques such as [22] can be employed on the CNF to efficiently recover an incomplete set of definitions. Given the set of such definitions, we can extract $\mathcal{I}$ and Φ with the desired properties. Our explicit definability-based identification focuses on two set of gates: AND ($\wedge$) and XOR ($\oplus$) gates. We detect AND gates of length two while we detect XOR gates of length up to five, both over literals rather than variables.

Our AND gate detection algorithm, inspired by [13] works up to length two only, since 2-long AND gates account for the overwhelming set of AND gates present in the benchmarks of interest. Since we focus on variables rather than literals, detection of OR gates is not required, since an OR gate is equivalent to an AND gate on the opposite literals due to De Morgan's laws [8].

In case of XOR gates, we rely on the bloom-filter based XOR recovery algorithm proposed by [29]. We instantiate this recovery algorithm to find XORs of length up to five in order to limit potential explosion of search space. Unlike AND gates, the identification of XOR equation does not put restrictions on the outputs and inputs. In particular, observe that an XOR $\ell_1 \oplus \ell_2 \oplus \ell_3 = 1$ can be rewritten in the following ways: (1) $\neg\ell_1 = \ell_2 \oplus \ell_3$, (2) $\neg\ell_2 = \ell_1 \oplus \ell_3$, and (3) $\neg\ell_3 = \ell_1 \oplus \ell_2$. Therefore, given an XOR of length k, we extract k definitions.

Algorithm 2 GreedyIndSearch(gates,$\mathcal{P}$)

```
1:  SortInc(P, incidence)                    ▷ Most likely dependent first
2:  for u ∈ P do                  ▷ Take most likely dependent variable
3:      for g ∈ gates[u] do
4:          OK ← True
5:          for v ∈ g.literals do
6:              if v ∉ P then              ▷ Could lead to cycle, skip
7:                  OK ← False
8:                  break
9:          if OK then
10:             P ← P \ u
11:             break
```

Irregular Gates. While finding AND and XOR gates syntactically is relatively straightforward, there are many possible other gates. To recover an irregular gate with output x, we can check the unsatisfiability of the formula $\varkappa = \{\varphi_x | \neg x \cup \varphi_{\neg x} | x\}$, as per [4]. If calling a SAT solver on $\varkappa$ returns UNSAT, we know that x is definable by the variables in $\varkappa$.

While computing the unsatisfiable subset of $\varkappa$ would allow us to recover the inputs of these gates (and use Algorithm 2), this would be expensive. Instead, we order literals in $\mathcal{P}$ according to their incidence, and run the irregular gate detection on them one by one, restricted to $\mathcal{P}$ as inputs. More specifically, to check if x is definable, we compute the formula ψ containing all clauses in φ that contain only variables from $\mathcal{P}$, and check whether $\{\psi_x | \neg x \cup \psi_{\neg x} | x\}$ is UNSAT. If it is, we can safely remove variable x from $\mathcal{P}$. Notice that computing φ is cheap given occurrence lists, maintained by all modern SAT solvers [10]. Given φ, computing ψ is straightforward via a check on all clauses' literals in φ. In order to reduce the computational overhead, we restrict the sub-solver to a limited number of conflicts (specifically, 100), to try to prove UNSAT.

The ExplicitSearch *algorithm.* Once the gates are recovered, we represent them in the obvious fashion by a graph, and compute the feedback vertex set of the graph. This latter is done by performing a greedy search to compute $W \cup \text{Root}(G)$ wherein W is a feedback vertex set of G. While ideally, we would like W to be a minimal feedback vertex set, we trade off the minimality for runtime performance. The key strategy is to sort the vertices according to the number of edges incident onto them; i.e., we observe that the higher the number of edges incident upon a vertex u, the higher likelihood of u to belong to $\mathcal{I}$. We present the pseudocode of the greedy search in Algorithm 2. The algorithm GreedyIndSearch relies on the array incidence that is computed for each variable u as the number of clauses containing u or $\neg u$ in φ.

4.2 Implicit Definability-based Identification

As noted in Section 2, Lagniez et al. observed that one can iteratively identify independent support. We first extend Lagniez et

[4] A possible method for computing definitions would be to rely on Slivovky's observation of extractions of definitions from the resolution proofs but such a technique is computationally expensive

Algorithm 3 ExplicitSearch($\varphi, \mathcal{P}$)

Step 1 Build a list of gate-definitions wherein for every gate g is represented as a tuple of the form $(\ell, \text{op}, \text{litList})$ such that $\ell = \text{op}(\text{litList})$, i.e., literal ℓ can be expressed as output of the gate corresponding to the operator op over the literal list litList.

Step 2 Construct a directed graph $G = (V, E)$ where there is a vertex $v \in V$ for every variable $x \in \mathcal{P}$, and for every gate $(\ell, \text{op}, \text{litList})$, we have an edge from variables in litList to the variable of ℓ. It is worth emphasizing that the gates are defined over litList.

Step 3 Compute feedback vertex set W of G in a greedy fashion and return $\mathcal{I} = W \cup \text{Root}(G)$ wherein $\text{Root}(G)$ corresponds to the set of vertices in G without any incoming edges.

al.'s proposal to handle projection set and present the resulting pseudocode, called SimpleSearch, in Algorithm 4.

Algorithm 4 SimpleSearch($\varphi, \mathcal{P}$)

```
1:  Y ← CreateNewVars(X); Z ← CreateNewVars(P)
2:  ψ ← φ(X) ∧ φ(X ↦ Y) ∧ ⋀_{i=1}^{|P|}(z_i → (x_i = y_i))
3:  solver.addConstraint(ψ)
4:  unknown ← {1, 2, ... |P|}; I ← ∅
                                   ▷ Sort most likely dependent last
5:  sortDesc(unknown, incidence)
6:  while unknown ≠ ∅ do
7:      assumps.clear()
                                   ▷ Take most likely dependent variable
8:      index ← unknown.pop()
9:      for j ∈ I do assumps.append(z_j)
10:     for j ∈ unknown do assumps.append(z_j)
11:     assumps.append(x_index)
12:     assumps.append(¬y_index)
13:     ret ← solver.solve(assumps)
14:     if ret ≠ UNSAT then I.append(index)
15: return I
```

SimpleSearch takes in a formula φ and a projection set $\mathcal{P}$, and returns the independent support $\mathcal{I}$. Without loss of generality, we assume that $\mathcal{P} = \{x_1, x_2, \ldots x_{|P|}\}$. The key idea is to construct ψ and perform iterative solve queries over ψ. The standard method is to use assumption-based framework where the solver is required to solve the formula under the set of assumptions expressed as assignment to variables. In SimpleSearch, we maintain two sets: unknown, the set of variables that are yet to be classified as dependent or independent and $\mathcal{I}$, the set of variables that we have classified as belonging to the independent support. As will be observed later, we would ideally like to sort the variables in such a way that the variables belonging to the independent support are queried at the very end.

Note that variable x_{index} can be defined in terms of unknown $\cup\, \mathcal{I}$ if and only if $\psi \wedge x_{\text{index}} \wedge \neg y_{\text{index}}$ is UNSAT under the assumption of setting all $z_i \in \text{unknown} \cup \mathcal{I}$ to True. Since some SAT calls may be very expensive, instead of invoking the solver to completion, we set a cutoff on the number of conflicts, and therefore, the solver may return SAT, UNSAT, or timeout. To account for timeout, we check, in line 14, whether ret ≠ UNSAT instead of checking ret = SAT.

We now seek to understand the key bottleneck for scalability of SimpleSearch: the call to SAT solver on line 13. To this end, we first seek to understand how modern CDCL-based SAT solvers implement the solve procedure.

Algorithm 5 IntegratedImplicit(φ, P)

```
1:  Y ← CreateNewVars(X); Z ← CreateNewVars(P)
2:  ψ ← φ(X) ∧ φ(X ↦ Y) ∧ ⋀_{i=1}^{|P|}(z_i → (x_i = y_i))
3:  solver.addConstraint(ψ)
4:  unknown ← {0, 1, 2, ... |P| − 1}; I ← ∅
5:  sortDesc(unknown, incidence)
6:  for i ∈ unknown do assumps.push_back(z_i)
7:  while True do
8:  start:
9:      branch ← None
10:     for i ← branch_depth; i < assumps.size() AND branch is
    None; i++ do
11:         lit ← assumps[i]; index ← i + 1
12:         if value(lit) is False then
13:             assumps.pop_back(); assumps.pop_back()
14:             if assumps.size() = I.size() then return I
15:             assumps.push_back(x_index)
16:             assumps.push_back(¬y_index)
17:             continue
18:         if value(lit) is True then
19:             new_decision_level()
20:             continue
21:         branch ← lit
22:     if branch is None then
23:         branch ← pick_branch()
24:         if branch is None OR conflict_limit() then
25:             assumps.pop_back(); assumps.pop_back()
26:             I.append(x_index)
27:             splice ← I.size()
28:             assumps.insert(splice, z_index)
29:             backtrack_until(splice)
30:             if assumps.size() = I.size() then return I
31:             assumps.push_back(x_index)
32:             assumps.push_back(¬y_index)
33:             go to start
34:     new_decision_level()
35:     enqueue(branch)
36: prop:
37:     ret ← propagate()
38:     if ret = conflict then
39:         analyze_conflict();
40:         backtrack_until(compute_backjump_level())
41:         go to prop
42:     if should_restart() then
43:         backtrack_until(assumps.size())
```

Observe that every invocation of solve on line 13 of SimpleSearch would require insertion of the set of assumptions of the size $|\mathcal{I} \cup \text{unknown}|$ and the resulting propagations. Since we invoke solve $|\mathcal{P}|$ times, the underlying solver must deal with $\frac{|\mathcal{P}|^2}{2}$ insertions and the corresponding propagations. To put this into perspective, if $|\mathcal{P}| = 50000$, then we have over a billion calls to propagate, a relatively expensive operation. At this point, observe that the variable appearing first in unknown is inserted and propagated for all except one solve call. Therefore, we focus on addressing the performance bottleneck via eliminating redundant work. To this end, the key idea is to pursue a tight integration of SimpleSearch and the solve algorithm, combining the two into a single algorithm, where solve is the algorithm as per the seminal MiniSat paper [10].

The pseudocode for this novel integrated approach, called IntegratedImplicit, is presented in Algorithm 5. We assume that φ is satisfiable, else $\mathcal{I} = \emptyset$ can be returned. Similar to SimpleSearch, we construct the formula ψ based on the input formula φ. The high-level structure of IntegratedImplicit is similar to SimpleSearch with crucial difference arising in the low-level technical details of how to handle the cases when UNSAT, SAT, or the timeout limit is reached.

We now focus on the UNSAT case, i.e., when x_{index} is shown to be dependent (lines 12– 17). The key observation is that whenever φ is satisfiable, then $\psi \wedge \bigwedge_{i=1}^{\text{index}} z_i$ is satisfiable. (Observe that $\{z_i\}_{i=1}^{\text{index}}$ is conjuncted via assumps). Therefore, if $\psi \wedge \bigwedge_{i=1}^{\text{index}} z_i \wedge x_{\text{index}} \wedge \neg y_{\text{index}}$ is unsatisfiable, then we need to only remove the three assumptions, namely $\{x_{\text{index}}, \neg y_{\text{index}}, z_{\text{index}}\}$ and do not need to backtrack to decision level zero.

In case of SAT (i.e., no more variables to branch on) or when the conflict limit is reached, we want to insert z_{index} into our assumps such that it is never popped during the rest of the execution. To this end, we insert z_{index} at the index determined by the current size of $\mathcal{I}$ and backtrack there, then continue running.

To summarize, IntegratedImplicit effectively avoids backtracking more than 3 levels except in case of (1) a regular conflict (2) restarting, or (3) an independent variable is found. In case of a restart, SAT solvers normally go back to decision level 0 but here, that would deterministically re-create what has already been decided and propagated until decision level assumps.size(), so we go back there instead. The usage of sorting of variables based on incidence, defined as the number of clauses containing the variable or its negation in φ as per [15] ensures that in practice, dependent variables (which are typically the vast majority of variables) are popped first. Therefore, in practice, we achieve a reduction from quadratic to linear in the number of propagation calls.

4.3 Arjun: Putting It All Together

Our proposed technique, Arjun, consists of combining the two phases ExplicitSearch and IntegratedImplicit, sequentially. As noted earlier, both GreedyIndSearch and IntegratedImplicit take a formula φ and a projection set $\mathcal{P}$ and return $(\varphi, \mathcal{I})$, where $\mathcal{I}$ is the independent support of $\mathcal{P}$. Observe that if $\mathcal{I}_1$ is independent support of $\mathcal{P}$ and $\mathcal{I}_2$ is independent support of $\mathcal{I}_1$, then $\mathcal{I}_2$ is independent support of $\mathcal{P}$. Therefore, the two phases can be combined in an arbitrary order as the output of one phase can be fed as the projection set for another phase. Our implementation of Arjun invokes ExplicitSearch before IntegratedImplicit.

Post-processing. In addition to IntegratedImplicit and ExplicitSearch, our implementation of Arjun involves a post-processing step focused on identifying *don't cares*. We focus on a simple syntactic check, i.e., for a variable x_i, we seek to identify whether for every clause C that contains literal x_i, there is also a clause C' containing $\neg x_i$ such that $C \setminus x_i = C' \setminus \neg x_i$, i.e., clauses C and C' are identical once x_i and $\neg x_i$ are removed from them respectively. For example, if the variable x_1 is only present in the following clauses: $\{x_1 \vee \neg x_2, \neg x_1 \vee \neg x_2, x_1 \vee x_4 \vee x_5, \neg x_1 \vee x_4 \vee x_5\}$, then x_1 is a *don't care*. Let the set of all such *don't cares* be M, then we can simply perform projected counting of the formula φ over the independent support $\mathcal{I} \setminus M$, and multiply the count by $2^{|M|}$ to get the correct count.

5 EMPIRICAL EVALUATION

We developed a prototype implementation of Arjun. The experiments were conducted on a high performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM, i.e 4GB limit per run. As both the tool[5] and the input instances [20] are available open-source, the below experiments are reproducible.

To evaluate the performance and the quality of independent support computed by Arjun, we conducted a comprehensive study on the state of the art counter ApproxMC4 and sampler UniGen3. ApproxMC4 is a highly competitive model counter, a version of which won the 2020 model counting competition in the projected counting track. The 2021 competition sought to focus on exact techniques and consequently, changed $\varepsilon = 0.1$ to $\varepsilon = 0.01$. Even then, ApproxMC4-based entry achieved 3rd place. As described during the competitive event of SAT 2021, had ε been set to 0.05, the ApproxMC4-based entry would have won the competition. All prior applications and benchmarking for approximation techniques have been presented with $\varepsilon = 0.8$ in the literature.

For our evaluation, we used 1896 benchmarks as released by Soos and Meel [20]. It comprises of a wide range of application areas including probabilistic reasoning, plan recognition, DQMR networks, ISCAS89 combinatorial circuits, quantified information flow, program synthesis, functional synthesis, logistics, and the like. The past few years have witnessed a surge of interest in projected counting and sampling; accordingly, 801 out of 1896 benchmarks specify a projection set. These benchmarks are tailored to counting and sampling, and are satisfiable. However, Arjun correctly handles UNSAT instances, returning an empty independent support, if it does not time out.

The prior state of the art approach, B + E, computes independent support only for the case when $\mathcal{P} = X$. On the other hand, MIS by [14] can compute independent support for an arbitrary $\mathcal{P}$ but is often significantly slower than B + E for the case when $\mathcal{P} = X$. Therefore, to ensure a comprehensive comparison, we experiment with both B + E and MIS. When we perform empirical evaluation of Arjun vis-a-vis B + E, we ignore the $\mathcal{P}$ supplied with the instance and instead set $\mathcal{P} = X$. In case of empirical evaluation with MIS, we use the $\mathcal{P}$ as supplied by the instances.

[5] CryptoMiniSat (https://github.com/msoos/cryptominisat) SHA1 rev. 5b1a027a2e08, ApproxMC (https://github.com/meelgroup/approxmc) SHA1 rev. 7eaca7c96d69, Arjun (https://github.com/meelgroup/arjun) SHA1 rev. 75b83f5428b7

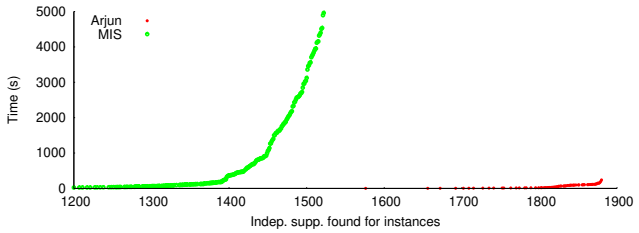

Figure 1: Arjun **vs.** MIS **independent support calculation times**

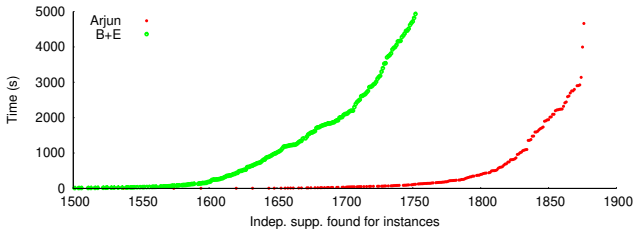

Figure 2: Arjun **vs** B + E **independent support calculation times. Here, the projection set was discarded for an apples-to-apples comparison. Both systems used a 500 conflicts/variable timeout**

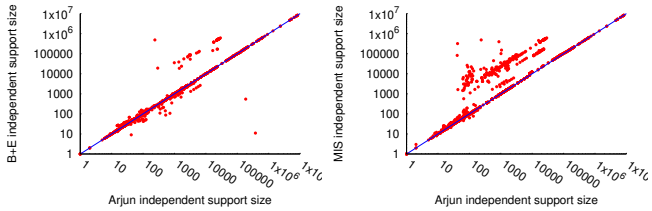

Figure 3: Computed independent support sizes

To understand the impact of independent support computation on ApproxMC, we cannot ignore $\mathcal{P}$ since under standard complexity theoretic assumptions, transformation to an equivalent instance without projection set (i.e., setting $\mathcal{P} = X$) would entail an exponential blow-up as such a transformation amounts to quantifier elimination. Therefore, our empirical evaluation performs comparisons of ApproxMC4 and UniGen3[6] preceded by Arjun and MIS respectively, owing to B + E's lack of support for projection. To put our performance improvements in perspective, we also evaluated the performance of ApproxMC3 and UniGen2 on our benchmarks. In line with prior studies, we set $\varepsilon = 0.8$ and $\delta = 0.2$ for all the versions of ApproxMC; in case of (all the versions of) UniGen, we set $\varepsilon = 16$.

Research Questions. Our empirical study sought to answer the following research questions: **RQ 1.** How does the runtime performance and the size of independent supports computed by Arjun compare via-a-vis to prior state of the approaches? **RQ 2.** How do different phases affect the runtime performance of Arjun? **RQ 3.** How does the augmentation of Arjun affect the runtime performance of hashing-based counting and sampling tools?

Summary of Results. Overall, we observe that Arjun significantly outperforms B + E and MIS in runtime performance. Furthermore, while we observe the critical importance of both phases ExplicitSearch and IntegratedImplicit, the empirical analysis shows that the incremental impact of IntegratedImplicit is higher than that of ExplicitSearch. We then observe that ApproxMC4 augmented with Arjun can count 1799 benchmarks, achieving an improvement of 576 benchmarks over ApproxMC4. Similarly, we observe that UniGen3 augmented with Arjun can sample 1372 benchmarks, achieving an improvement of 335 benchmarks within the same time limit of 5000s. Furthermore, we observe a significant improvement in runtime performance as well, in particular, the PAR-2 scores for ApproxMC4 and UniGen3 augmented with Arjun are 575 and 2861 while the PAR-2 scores for ApproxMC4 and UniGen3 are 3717 and 4678, respectively.

5.1 Comparison of Arjun with MIS and B + E

We present the runtime performance of Arjun vis-a-vis MIS via cactus plot in Figure 1. Observe that while Arjun could compute independent support for 1880 instances, MIS could do the same for only 1522 instances within the same time limit. To illustrate the runtime performance difference, Arjun computed independent support for the same number of instances (i.e. 1522) in under 0.60s that MIS took 5000s.

Even though from complexity theoretic viewpoint, we cannot ignore $\mathcal{P}$, we sought to further extensively analyze performance of Arjun. To this end, we performed empirical evaluation vis-a-vis B + E by disregarding the user-provided projection set and instead setting $\mathcal{P} = X$ due to B + E's inability to perform independent support computation for a non-trivial $\mathcal{P}$. We present the runtime performance cactus plot in Figure 2. While it took 4932 seconds for B + E to compute the independent support of all 1752 instances it could compute, for Arjun it only took 117 seconds to do the same.

At this point, one may wonder about the size of independent supports computed by Arjun, MIS, and and B + E. To this end, we present scatterplot of the size of independent support found by Arjun, MIS, and B + E in Figure 3. For instances where a tool could not compute the independent support, we show the default projection set size of the instance.

5.2 Impact of ExplicitSearch and IntegratedImplicit

To understand the impact of ExplicitSearch and IntegratedImplicit, we present the runtime performance of different versions of Arjun in Figure 4. The curve "Arjun w/o IntegratedImplicit" refers to the version of Arjun with SimpleSearch instead of IntegratedImplicit, while "Arjun w/o ExplicitSearch" refers to the version of Arjun without explicit definability-based identification. We make two observations: (1) both phases play an important role in the performance of the tool, and (2) the impact of IntegratedImplicit is higher than that of ExplicitSearch.

[6]Latest recommended versions of ApproxMC and UniGen

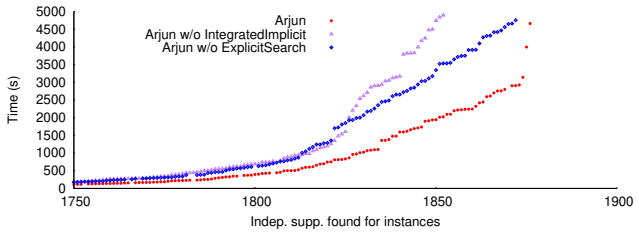

Figure 4: Impact of IntegratedImplicit **and** ExplicitSearch **on** Arjun

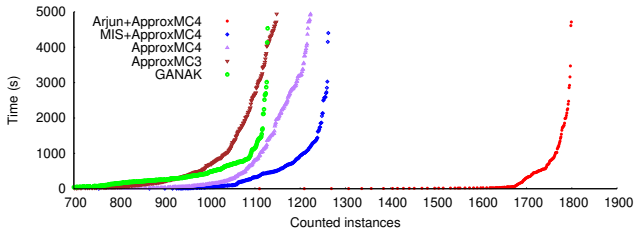

Figure 5: GANAK **state of the art exact model counter [26] for reference,** ApproxMC, Arjun+ApproxMC, **and** MIS+ApproxMC **projected model counting time**

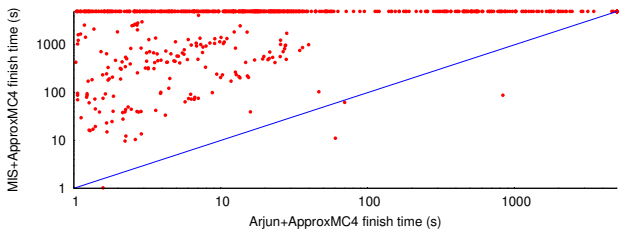

Figure 6: Arjun+ApproxMC4 **vs.** MIS+ApproxMC4 **time**

5.3 Impact of Arjun on ApproxMC4

The cactus plot of ApproxMC3, ApproxMC4, MIS+ApproxMC4, and Arjun+ApproxMC4 (cumulative time) are on Figure 5. Here, Arjun or MIS was ran first on the benchmark, and the resulting independent support was provided to ApproxMC4. Arjun+ApproxMC4 could count 1799 instances while ApproxMC4 and MIS+ApproxMC4 could only count 1223 and 1262 instances, respectively within the same 5000s timeout. Overall, we observe an improvement of 576 instances. To put this into context, we also plot the curve corresponding to ApproxMC3, which demonstrates significant improvement Arjun provides relative to ApproxMC4 vs ApproxMC3. Solving speed improvement is substantial: with a timeout of only 2.24 seconds, Arjun+ApproxMC4 could solve as many instances as ApproxMC4 on its own under 5000 seconds.

To further compare the runtime on a per instance basis, we present the scatter plot of Arjun+ApproxMC4 in comparison with MIS+ApproxMC4 in Figure 6. We observe that on a per instance basis Arjun+ApproxMC4 is almost always faster than MIS+ApproxMC4.

This demonstrates the importance of efficient computational technique that can find small independent support. Recall that ApproxMC4 adds XORs over the independent support in order to perform approximate counting. Hence, a smaller independent support leads to faster propagation and on average smaller conflict clauses.

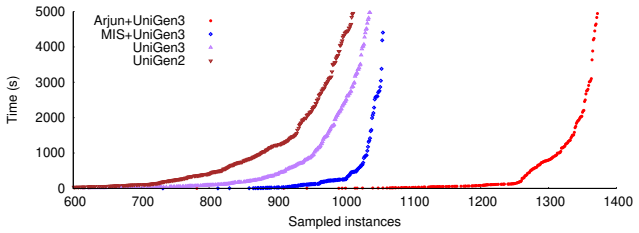

Figure 7: Runtime performance of UniGen, MIS+UniGen, **and** Arjun+UniGen

5.4 Impact of Arjun on UniGen3

Figure 7 shows the cactus plot of UniGen2, UniGen3, MIS+UniGen3, and Arjun+UniGen3 (cumulative time). Overall, Arjun+UniGen3 samples 1372 instances, 316 more than MIS+UniGen3 within the same 5000s timeout. To put such a performance improvement into context, observe that the difference between UniGen2 and UniGen3 in terms of number of sampled instances is significantly less than MIS+UniGen3 vs Arjun+UniGen3.

6 CONCLUSION

In this paper, we focused on the problem of computation of independent support for a given formula owing to its importance for hashing-based counting and sampling techniques. Our algorithmic framework, Arjun, consists of two phases that seek to take advantage of implicit and explicit definability. The extensive empirical evaluation shows that Arjun achieves significant performance improvement over prior state of the art. Furthermore, augmenting with Arjun the state of the art hashing-based counter ApproxMC4 and sampler UniGen3 leads to significant performance improvements.

Acknowledgments. This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] , Ministry of Education Singapore Tier 2 grant MOE-T2EP20121-0011, Ministry of Education Singapore Tier 1 Grant [R-252-000-B59-114], and Amazon Research Award. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg

REFERENCES

[1] Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264 (2019)
[2] Balyo, T., Heule, M.J.H., Järvisalo, M.: Preface. In: Balyo, T., Heule, M.J.H., Järvisalo, M. (eds.) Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions. pp. 3–3 (2017)
[3] Beth, E.W.: On Padoa's method in the theory of definition. In: Journal of Symbolic Logic 21. pp. 194–195. No. 2 (1956)

[4] Biere, A.: Lingeling, Plingeling and Treengeling entering the SAT Competition 2013. In: Balint, A., Belov, A., Heule, M.J.H., Järvisalo, M. (eds.) Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions. pp. 51–52 (2013)

[5] Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of SAT witnesses. In: Proc. of CAV. pp. 608–623 (2013)

[6] Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Proc. of CP. pp. 200–216 (2013)

[7] Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proc. of DAC. pp. 1–6 (2014)

[8] De Morgan, A.: Trigonometry and Double Algebra. Taylor, Walton, and Malbery, London (1849)

[9] Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proc. of AAAI (2017)

[10] Eén, N., Sörensson, N.: An extensible SAT-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)

[11] Ermon, S., Gomes, C., Sabharwal, A., Selman, B.: Low-density parity constraints for hashing-based discrete integration. In: International Conference on Machine Learning. pp. 271–279. PMLR (2014)

[12] Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: Short XORs for model counting: from theory to practice. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 100–106. Springer (2007)

[13] Iser, M.: Recognition and Exploitation of Gate Structure in SAT Solving. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2020)

[14] Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints pp. 1–18 (2015). https://doi.org/10.1007/s10601-015-9204-z

[15] Lagniez, J.M., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: IJCAI. pp. 751–757 (2016)

[16] Lagniez, J.M., Lonca, E., Marquis, P.: Definability for model counting. Artificial Intelligence **281**, 103229 (2020)

[17] Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1), 1–33 (2008)

[18] Malik, S., Zhao, Y., Madigan, C.F., Zhang, L., Moskewicz, M.W.: Chaff: Engineering an efficient SAT solver. Design Automation Conference pp. 530–535 (2001). https://doi.org/http://doi.ieeecomputersociety.org/10.1109/DAC.2001.935565

[19] Meel, K.S., Akshay, S.: Sparse hashing for scalable approximate model counting: theory and practice. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 728–741 (2020)

[20] Meel, K.S., Soos, M.: Model counting and uniform sampling instances. https://doi.org/10.5281/zenodo.3793090 (May 2020). https://doi.org/10.5281/zenodo.3793090

[21] Naveh, Y., Emek, R.: Random Stimuli Generation for Functional Hardware Verification as a CP Application. In: Proc. of CP. Lecture Notes in Computer Science, vol. 3709, pp. 882–882. Springer (2005)

[22] Ostrowski, R., Grégoire, É., Mazure, B., Sais, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Proc. of CP. Lecture Notes in Computer Science, vol. 2470, pp. 185–199. Springer (2002)

[23] Padoa, A.: Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une theorie déductive quelconque. In: Bibliothèque du Congrès international de philosophie. vol. 3, pp. 309–365 (1901)

[24] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. pp. 294–299. Springer (2007)

[25] Sashittal, P., El-Kebir, M.: Sharptni: counting and sampling parsimonious transmission networks under a weak bottleneck. bioRxiv p. 842237 (2019)

[26] Sharma, S., Roy, S., Soos, M., Meel, K.S.: Ganak: A scalable probabilistic exact model counter. In: IJCAI. vol. 19, pp. 1169–1176 (2019)

[27] Slivovsky, F.: Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In: International Conference on Computer Aided Verification. pp. 508–528. Springer (2020)

[28] Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In: International Conference on Computer Aided Verification. pp. 463–484. Springer (2020)

[29] Soos, M., Meel, K.S.: Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting. In: Proc. of the AAAI (2019)

[30] Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing **8**(3), 410–421 (1979)