On computing Minimal Independent Support and its applications to sampling and counting [⋆]

Alexander Ivrii[1], Sharad Malik[2], Kuldeep S. Meel[3], and Moshe Y. Vardi[3]

[1] IBM Research Lab, Haifa, Israel
[2] Princeton University
[3] Department of Computer Science, Rice University

Abstract. Constrained sampling and counting are two fundamental problems arising in domains ranging from artificial intelligence and security, to hardware and software testing. Recent approaches to approximate solutions for these problems rely on employing SAT solvers and universal hash functions that are typically encoded as XOR constraints of length $n/2$ for an input formula with n variables. As the runtime performance of SAT solvers heavily depends on the length of XOR constraints, recent research effort has been focused on reduction of length of XOR constraints. Consequently, a notion of *Independent Support* was proposed, and it was shown that constructing XORs over independent support (if known) can lead to a significant reduction in the length of XOR constraints without losing the theoretical guarantees of sampling and counting algorithms. In this paper, we present the first algorithmic procedure (and a corresponding tool, called MIS) to determine minimal independent support for a given CNF formula by employing a reduction to group minimal unsatisfiable subsets (GMUS). By utilizing minimal independent supports computed by MIS, we provide new tighter bounds on the length of XOR constraints for constrained counting and sampling. Furthermore, the universal hash functions constructed from independent supports computed by MIS provide two to three orders of magnitude performance improvement in state-of-the-art constrained sampling and counting tools, while still retaining theoretical guarantees.

1 Introduction

Constrained sampling and counting are two fundamental problems arising in domains such as artificial intelligence, hardware and software testing and the like. In constrained sampling, the task is to sample randomly, subject to a given weight function, from the set of solutions of input constraints. In constrained counting, the task is to count the total weight, subject to a given weight function, of the set of solutions of input constraints. Both problems have numerous applications, including probabilistic reasoning, machine learning, planning, statistical physics, inexact computing, and constrained-random verification[2,11,27,30,33].

[⋆] The author list has been sorted alphabetically by last name; this should not be used to determine the extent of authors' contributions.

For example, probabilistic inference over graphical models can be reduced to constrained counting for propositional formulas [2,31]. In addition, approximate probabilistic reasoning relies heavily on sampling over high-dimensional probabilistic spaces encoded as a set of constraints [18]. The problems of constrained sampling and counting can be viewed as aspects of one of the most fundamental problem in artificial intelligence, which is to explore the structure of the solution space of a given set of constraints [29].

Constrained sampling and counting are known to be computationally hard [32,19]. To address the intractability barriers, approximations of these problems have been pursued. Of these, Monte Carlo Markov Chain (MCMC) algorithms, such as those based on simulated annealing and Metropolis-Hastings and variational methods have been extensively studied in this context, but these algorithms settle for very weak guarantees (such as eventual asymptotic convergence) or do not provide formal guarantees at all [20]. Interval propagation and random seeding of underlying constraint solvers have also failed to provide theoretical guarantees and often generate highly irregular distributions or compute inexact counts [20].

To counter the scalability and approximation challenges for sampling and counting, new techniques based on universal hashing were proposed in [7,8,9,6,14]. These techniques combine the classical technique of universal hashing with the recent advancements in combinatorial reasoning techniques such as Boolean satisfiability (SAT). The key idea in these techniques is to employ universal hashing to partition the solution space into "small" cells with each cell containing a relatively small expected sum of weight of solutions. The counting algorithms then typically invoke combinatorial search tools such as CryptoMiniSAT to repeatedly compute the size of sufficiently many random cells (to achieve the desired confidence) and to return the estimate as the number of cells multiplied by the median of the sizes returned by invocations. Similarly, one can sample almost uniformly by first choosing a random cell and then sampling uniformly inside the cell [9]. The use of universal hash functions allows these techniques to provide theoretical guarantees of (ε, δ)-approximation, which is far stronger than the previous state-of-the-art approaches.

The recent hashing-based techniques predominately employ *3-universal hash functions*,[4] which typically consist of a conjunction of parity constraints, i.e. XOR, each of average density of 1/2. As a result, a cell is represented as a conjunction of input constraints and XOR constraints. Since combinatorial reasoning tools are invoked to search for solutions of the conjunction of input constraints and XOR-based universal hash functions, recent investigations have focused on determining the relationship between runtime performance of combinatorial search and features of XOR-constraints. It has been observed that lower density XORs are easy to reason in practice and runtime performance of solvers greatly enhances with the decrease in the density of XOR-constraints [16]. This has led to recent work focused on designing hash functions with low density XOR-constraints [12] but such hash functions provide very weak guarantees of universality that are not adequate for hashing-based techniques for sampling.

[4] defined formally in Section 2

Recently, Chakraborty et al. introduced the notion of an *independent support* of a Boolean formula [9]: a subset of variables whose values uniquely determine the values of the remaining variables in any satisfying assignment to the formula, and showed that XOR-based 3-universal hash functions can be constructed by picking variables from the independent support alone. The importance of this observation comes from the fact that for many important classes of problems the size of an independent support is typically one to two orders of magnitude smaller than the number of all variables, which in turn leads to XOR constraints of typical density of 1/200 to 1/20, i.e. one to two orders of magnitude smaller than that of the traditional hash functions. We emphasize that unlike recent work of Ermon et al., these hash functions still preserve the strong guarantees of $3-$*universality* and therefore can be used as replacement for traditional hash functions in recent hashing-based techniques for sampling and counting such as UniWit [7], ApproxMC [8], WeightMC, WeightGen [6], PAWS [13]. The authors, however, left open the question of an efficient algorithmic procedure for determining a "small" independent support of a Boolean formula (for example, minimal or minimum-sized).

The variables that are not part of an independent support can be considered as *redundant*, and there is extensive research related to redundancy in propositional logic in general [21,3]. In our context, a particular problem of importance is that of computing a concise reason of inconsistency of an over-constrained Boolean formula. Significant recent research focuses on efficiently computing a *minimal unsatisfiable subformula* (MUS) of a Boolean formula [25] and its extension on computing a *group-oriented* (also called *high-level*) minimal unsatisfiable subformula (GMUS) of an explicitly partitioned Boolean formula [23,26]. In addition, there are highly optimized algorithms and off-the-shelf implementations for computing MUSes and GMUSes, such as MUSer2 [4]. Even more recent research focuses on computing a *smallest* (i.e., minimum-sized) MUS of a Boolean formula (SMUS) [22,1], which in general is a significantly more computationally-intensive task. Similarly, one can consider a smallest GMUS of an explicitly partitioned Boolean formula (SGMUS). The tool Forqes described in [1] can compute SMUSes and SGMUSes.

The primary contribution of this paper is, to the best of our knowledge, the first algorithm to determine minimal and minimum independent supports. The key idea of this algorithmic procedure is the reduction of the problem of minimizing an independent support of a Boolean formula to (S)GMUS. In this reduction, each independent subset of variables naturally corresponds to an unsatisfiable subformula of the total formula, and in particular the problems of finding a minimal independent support, or a minimum-sized independent support, or all minimal or minimum-sized independent supports, can be naturally translated to the corresponding problems in the MUS framework. For future reference, we denote by MIS the tool that computes a minimal independent support of a Boolean formula by employing the above translation and MUSer2, and we denote by SMIS the tool that computes a minimum independent support that uses Forqes instead.

To illustrate practical gains, we used MIS to compute a minimal independent support for each of the benchmarks, and we augmented state of the art sampler, UniGen2, and model counter, ApproxMC, with the new hashing scheme that uses the computed minimal independent subsets. Experimental comparison over a wide suite of benchmarks demonstrate that new hashing scheme results in improvement of runtime performance of UniGen2 and ApproxMC by two to three orders of magnitude. It is worth noting that runtime improvement for ApproxMC and UniGen2 comes at no cost; ApproxMC and UniGen still provide the same strong theoretical guarantees.

An experimental comparison of MIS and SMIS highlights an important trade-off between the performance and the sizes of computed independent supports. In particular, while MIS scales to larger formulas, SMIS computes even smaller independent supports for a subset of benchmarks that are within its reach.

The remainder of the paper is organized as follows. We introduce notation and preliminaries in Section 2, and discuss related work in Section 3. We present our main technical contribution – identification of a minimal set of independent variables – in Section 4. We augment state of art sampling and model counting tools with the hashing scheme based on independent variables and demonstrate effectiveness of the new hashing scheme over a wide set of benchmarks in Section 5. Finally, we discuss future work and conclude in Section 6.

2 Preliminaries

Let F denote a Boolean formula in conjunctive normal form (CNF), and let X be the set of variables appearing in F. The set X is called the *support* of F. We also use $\mathsf{Vars}(F)$ to denote the support of F. Given a set of variables $S \subseteq X$ and an assignment σ of truth values to the variables in X, we write $\sigma|_S$ for the projection of σ on S. A *satisfying assignment* or *witness* of F is an assignment that makes F evaluate to true. We denote the set of all witnesses of F by R_F and the projection of R_F on S by $R_{F|S}$. For notational convenience, whenever the formula F is clear from the context, we omit mentioning it.

Let $\mathcal{I} \subseteq X$ be a subset of the support such that if two satisfying assignments σ_1 and σ_2 agree on I, then $\sigma_1 = \sigma_2$. In other words, in every satisfying assignment, the truth values of variables in I uniquely determine the truth value of every variable in $X \setminus \mathsf{I}$. The set $\mathcal{I}$ is called an *independent support* of F, and $\mathcal{D} = X \setminus \mathcal{I}$ is referred to as *dependent support.* Note that there is a one-to-one correspondence between R_F and $R_{F|I}$. There may be more than one independent support: $(a \vee \neg b) \wedge (\neg a \vee b)$ has three, namely $\{a\}$, $\{b\}$ and $\{a, b\}$. Clearly, if $\mathcal{I}$ is an independent support of F, so is every superset of $\mathcal{I}$.

A special class of hash functions, called r-*universal* hash functions, play a crucial role in our work. Let n, m and r be positive integers, and let $H(n, m, r)$ denote a family of r-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} H(n, m, r)$ to denote the probability space obtained by choosing a hash function h uniformly at random from $H(n, m, r)$. The property of r-universality guarantees that for all $\alpha_1, \ldots, \alpha_r \in \{0,1\}^m$ and for all distinct $y_1, \ldots, y_r \in$

$\{0,1\}^n$, $\mathsf{Pr}\left[\bigwedge_{i=1}^{r} h(y_i) = \alpha_i \ : h \xleftarrow{R} H(n,m,r)\right] = 2^{-mr}$. We use a particular class of such hash functions, denoted by $H_{xor}(n,m)$, which is defined as follows. Let $h(y)[i]$ denote the i^{th} component of the vector $h(y)$. This family of hash functions is then defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \cdot y[k]), a_{i,k} \in \{0,1\}, 1 \leq i \leq m, 0 \leq k \leq n\}$, where $\oplus$ denotes the XOR operation. By choosing values of $a_{i,k}$ randomly and independently, we can effectively choose a random hash function from $H_{xor}(n,m)$. It was shown in [28] that this family is 3-universal.

2.1 Group-oriented Unsatisfiable Subformulas and Subsets

In the problem of *group-oriented minimization* of unsatisfiable subsets [23,26], we are given an unsatisfiable formula Ψ of the form $\Psi = H_1 \wedge \cdots \wedge H_m \wedge \Omega$, and the task is to find a subset $\{H_{i_1}, \ldots, H_{i_k}\}$ of $\{H_1, \ldots, H_m\}$ so that $H_{i_1} \wedge \cdots \wedge H_{i_k} \wedge \Omega$ remains unsatisfiable. The subformulas $H_1, \ldots, H_m$ are called *groups* (or *high-level constraints*) and Ω is called the *remainder*. The remainder plays a special role – it consists of *non-interesting* constraints that do not need to be minimized and are always part of the formula.

If $H_{i_1} \wedge \cdots \wedge H_{i_k} \wedge \Omega$ is unsatisfiable, we say that $\{H_{i_1}, \ldots, H_{i_k}\}$ is a (group-oriented) unsatisfiable subset, or equivalently that $H_{i_1} \wedge \cdots \wedge H_{i_k} \wedge \Omega$ is an unsatisfiable subformula of Ψ. In addition, when $\{H_{i_1}, \ldots, H_{i_k}\}$ is *minimal* (removal of any H_{i_j} renders the formula satisfiable), we say that $\{H_{i_1}, \ldots, H_{i_k}\}$ is a *group-oriented minimal unsatisfiable subset* (GMUS), or equivalently that $H_{i_1} \wedge \cdots \wedge H_{i_k} \wedge \Omega$ is a minimal unsatisfiable subformula of Ψ. (If $\{H_{i_1}, \ldots, H_{i_k}\}$ is of minimum size, that is there is no smaller unsatisfiable subset, then we call it a *minimum* unsatisfiable subset (SGMUS).) A variety of highly optimized tools for computing GMUSes is available; in this paper we use MUSer2 [4]. To compute SGMUSes, we use Forqes [1].

3 Related Work

Since the notion of independent support was introduced by Chakraborty et al. in the context of designing efficient universal hash functions [9], we briefly review universal hash functions for constrained sampling and counting. The concept of universal hash functions was first proposed by Carter and Wegman in [5]. This led to seminal work by Bellare, Goldreich and Petrank, who proposed a $\mathsf{BPP}^{\mathsf{NP}}$ algorithm (referred to by BGP after the authors' initials), based on $n-$universal hashing to sample SAT witnesses uniformly. The requirement of n–universality, however, prohibited practical usage of the BGP algorithm. Building on the BGP algorithm, Chakraborty et al. proposed the UniWit [7], and subsequently UniGen [9] algorithms that sample SAT witnesses *almost* uniformly and require only 3-universal hashing [9]. The hashing-based techniques proposed in UniGen have also been extended to model counting, resulting in approximate model counters such as ApproxMC [8], WeightMC [6], WISH [14].

These hashing-based techniques crucially rely on the ability of combinatorial solvers to solve propositional formulas represented as conjunction of input formulas and 3-universal hash functions (typically H_{xor}), thus translating to *hybrid* formulas, which are conjunctions of CNF and XOR clauses. Therefore, a key challenge is to further speed up search for the solutions of such hybrid formulas. The XOR clauses invoked here are *high-density* clauses, consisting typically of $n/2$ variables, where n is the total number of atomic propositions. Experience has shown that the high density of the XOR clauses plays a major role in making solving hybrid formulas a hard problem.

Recently, Ermon et al. introduced a technique based on low-density XOR constraints, which provides weaker guarantee than 2–universality, but was shown to be sufficient for approximate counting [12]. Constrained sampling, however, requires the stronger guarantee of 3–universality [9], and therefore the low-density approach fails to address the problem of high-density XOR constraints, which motivated Chakraborty et al. to propose the notion of independent support I and show that H_{xor} constructed over I does provide 3–universality. While it is possible in many cases to obtain over-approximation of I by examining the source domain from which the problem is derived, the work in [9] does not provide an algorithmic approach for determining I, and experience has shown that the manual approach is error prone.

The notion of independent support is closely related to the concept of functional dependency in the context of relational databases; it is essentially equivalent to the concept of a *key* in a relation [24]. The difference is that in the context of relational databases, relations are represented *explicitly*, while here the relation R_F is represented *implicitly* by means of the formula F. Thus, algorithmic techniques from relational-database theory do not scale to the setting considered here. In the context of combinational logic circuits, there has been some work that constructs a logic circuit whose Tseitin-encoding corresponds to the given Boolean formula [15]. The primary inputs of this constructed circuit form the independent support of the given Boolean formula. This construction is based on pattern matching the formula to find sub-formulas corresponding to commonly used gates. This technique is not guaranteed to be complete and unlikely to succeed if the formulas did not originate from combinational circuits.

4 Computing Minimal/Minimum Independent Supports

In this section, we first discuss how computation of minimal/minimum independent supports can be reduced to computation of minimal/minimum unsatisfiable subsets. Building on our reduction, we propose the first algorithmic procedure, MIS, to compute a minimal independent support for a given formula. We then discuss how MIS can make efficient usage of information from users. We also discuss a variant SMIS that computes a minimum independent support. Finally, we discuss how minimal and minimum independent supports computed by MIS and SMIS can be applied to hashing-based approximate techniques for counting and sampling.

4.1 Reduction to Group-oriented Minimal Unsatisfiable Subsets

For a given Boolean formula F and $S \subseteq X$, we know that S is an independent support of F whenever every two satisfying assignments σ_1, σ_2 to F that agree on S, must be identical. We formalize this as follows. We introduce additional variables $Y = \{y_1, \ldots, y_n\}$, and let $F(y_1, \ldots, y_n)$ be obtained from $F(x_1, \ldots, x_n)$ by replacing every occurrence of a variable in X by the corresponding variable in Y. The definition of independence is captured by the following formula:

$$F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i \in Ind(S)} (x_i = y_i) \implies \bigwedge_{j \in Ind(X \setminus S)} (x_j = y_j),$$

where $Ind(S)$ and $Ind(X \setminus S)$, respectively, denote the index sets of S and $X \setminus S$. Since it obviously holds that $\bigwedge_{i \in Ind(S)} (x_i = y_i) \Rightarrow \bigwedge_{i \in Ind(S)} (x_i = y_i)$, we can replace the right-hand side of the above formula by $\bigwedge_{j \in Ind(X)} (x_j = y_j)$. Finally, define the Boolean function $Q_{F,S}(x_1, \ldots, x_n, y_1, \ldots, y_n)$ by

$$Q_{F,S} = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i \in Ind(S)} (x_i = y_i) \wedge \neg \left(\bigwedge_{j \in Ind(X)} (x_j = y_j) \right).$$

Proposition 1. *S in an independent support for F if and only if $Q_{F,S}$ is unsatisfiable.*

From Proposition 1 we obtain the following upper bound.

Theorem 1. *The problem of deciding whether S is a minimal independent support of F is in $\mathsf{D^P}$, where $\mathsf{D^P} = \{A - B | A, B \in \mathsf{NP}\}$.*

Proof. Checking that S is independent support of F is reducible to unsatisfiability of $Q_{F,S}$, which is in co-NP. To check minimality, we can select each variable $x \in S$ and check that $Q_{F,S-\{x\}}$ is satisfiable.

We offer the following lower bound conjecture:

Conjecture 1. The problem of deciding whether S is a minimal independent support of F is $\mathsf{D^P}$-complete.

Proposition 1 leads to algorithms for computing a minimal independent support of F. One possible approach is to start with $S = X$ and the obviously unsatisfiable formula $Q_{F,X}$, and then remove variables x_i from S (corresponding to conjuncts $x_i = y_i$ in $Q_{F,S}$) as long as $Q_{F,S}$ remains unsatisfiable. Instead, we observe that the problem of minimizing independent support can be restated as the problem of minimizing unsatisfiable subsets, and hence we can benefit from the full variety of different algorithms and various important optimizations developed in the latter context. We now pursue this direction.

Using notation from Section 2.1, define $H_1, \ldots, H_n$ and Ω as follows:

$$H_1 = \{x_1 = y_1\}, \ \ldots, \ H_n = \{x_n = y_n\},$$

$$\Omega = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigvee_{i \in Ind(X)} (x_i \neq y_i).$$

To obtain a CNF representation, suppose that the original formula F is given in CNF. Then we let $H_i = \{(\neg x_i \vee y_i) \wedge (x_i \vee \neg y_i)\}$ for $i = 1, \ldots, n$. For Ω, the terms $F(x_1, \ldots, x_n)$ and $F(y_1, \ldots, y_n)$ are already in CNF. To encode $\bigvee_{i=1}^{n}(x_i \neq y_i)$, we introduce additional variables $b_1, \ldots, b_n$, add clauses $(\neg x_i \vee \neg y_i \vee b_i)$, $(x_i \vee y_i \vee b_i)$ for $i = 1, \ldots, n$, and add the clause $(\neg b_1 \vee \cdots \vee \neg b_n)$.

The following proposition follows immediately from the construction and Proposition 1:

Proposition 2. *The formula $H_1 \wedge \cdots \wedge H_n \wedge \Omega$ is unsatisfiable. Moreover, for a subset $S \subseteq X$: S is an independent support of F if and only if $\{H_i | i \in Ind(S)\}$ is a group-oriented unsatisfiable subset of $\{H_1, \ldots, H_n\}$.*

It immediately follows that problems of computing independent support can be reduced to analogous problems of finding group oriented unsatisfiable subsets. Specifically, computing a minimal independent support can be reduced to computing a minimal unsatisfiable subset; computing a minimum independent support can be reduced to computing a minimum unsatisfiable subset; computing all minimal independent supports can be reduced to computing all minimal unsatisfiable subsets; and so on.

4.2 Handling Under- and Over- Approximations

In Section 4.3 we describe a light-weight technique for detecting a set of variables that is dependent on the remaining variables in the formula, thus allowing us to restrict the search for a minimal independent support by excluding the dependent variables. Furthermore, in some of our applications (see Section 4.6), the user has the additional freedom to specify which variables should or should not be in the independent support. In both cases, we can think of the set of variables that should to be included as specifying an *under-approximation* of the independent support, and we can think of complement of the set of variables that should be excluded as specifying an *over-approximation* of the independent support.

Due to these considerations, we introduce the following extension of the independent-support problem. Let $U \subseteq V \subseteq X$ and suppose that V is an independent support of F. Let us say that an independent support of F relative to an *under-approximation* U and an *over-approximation* V is a set S such that $U \subseteq S \subseteq V$ and S is an independent support of F. Further, let us say that a *minimal independent support of F relative to U and V* is a minimal S with these properties. Note that S does not need to be a minimal independent support of F (as U itself might have dependent variables). Also note the explicit requirement that V is an independent support (if V is not an independent support, then no subset of V is an independent support).

The reduction to group-oriented unsatisfiable subset described in Section 4.1 can be easily extended to handle this more general problem. Given F, U and V as above, let $H_i = \{x_i = y_i\}$ for $i \in Ind(V \setminus U)$, and let $\Omega = F(x_1, \ldots, x_n) \wedge F(y_1, \ldots, y_n) \wedge \bigwedge_{i \in Ind(U)}(x_i = y_i) \wedge \bigvee_{i \in Ind(X)}(x_i \neq y_i)$.

Proposition 3. *The following statements are true:*

1. *The formula $\Omega \wedge \bigwedge_{i \in Ind(V \setminus U)} H_i$ is unsatisfiable.*
2. *For a subset $W \subseteq V \setminus U$: $\{H_i \mid i \in Ind(W)\}$ is a group-oriented unsatisfiable subset of $\{H_i \mid i \in Ind(V \setminus U)\}$ if and only if $U \cup W$ is an independent support of F relative to U and V.*
3. *$\{H_i \mid i \in Ind(W)\}$ is a minimal group-oriented unsatisfiable subset of $\{H_i \mid i \in Ind(V \setminus U)\}$ if and only if $U \cup W$ is a minimal independent support of F relative to U and V.*

We, henceforth, denote this reduction as $\mathsf{TranslateToGMUS}(F, U, V)$. Note that when $U = \emptyset$ and $V = X$ the definition of an independent support relative to U and V corresponds to the standard definition of independent support, and $\mathsf{TranslateToGMUS}(F, U, V)$ coincides with the reduction given in Section 4.1. In what follows, we omit "relative to an under-approximation U" when $U = \emptyset$, and we omit "relative to an over-approximation V" when $V = X$.

4.3 Exploiting Local Dependencies

In various important contexts, a variable $x \in X$ can be shown to be dependent on other variables, either purely syntactically or by analyzing only a small subset of all clauses. An especially important case is when the formula F encodes a circuit, in which case many variables can be detected to be dependent simply from their defining clauses.

Example 1. Suppose that F contains the following clauses (among others): $(\neg x \vee y \vee b), (x \vee \neg y), (x \vee \neg b)$. It can be readily seen that in every satisfying assignment to F we have that $x = y \vee b$, and so x is dependent on $\{y, b\}$.

Intuitively, the variables that are locally dependent on other variables do not need be considered for independent support. We need, however, to avoid *cyclic reasoning*, such as when $F := (\neg x \vee y) \wedge (x \vee \neg y)$, x depends on y and that y also depends on x.

Algorithm 1 $\mathsf{FindLocalDependencies}$(F, V)

```
Input: CNF formula $F$; set $V \subseteq X$
Output: A subset $Z \subseteq V$ of dependent variables.
1: $Z = \emptyset$
2: for $x \in V$ do
3:     $G = \mathsf{SelectLocalClauses}(F, x)$
4:     $W = \mathsf{Vars}(G)$ /*$\mathsf{Vars}(G)$ denotes the support of $G$ */
5:     if $Q_{G, W \setminus \{x\}}$ is UNSAT then
6:         $Z = Z \cup \{x\}$
7:         $F = F \setminus G$
8: return $Z$
```

We propose Algorithm 1 to detect a set of non-cyclic locally dependent variables. The algorithm accepts a formula F in CNF and a set V of candidate variables to consider, and returns a set $Z \subseteq V$ of variables that are (non-cyclically) dependent on the remaining variables. Initially, Z is empty. In the algorithm we iteratively select a variable $x \in V$ and call SelectLocalClauses to select a set of clauses of F "around" x. These should include at least all the clauses of F involving x, but more generally can correspond to a larger neighborhood of x in the *primal graph* (the graph with vertexes $\mathsf{Vars}(F)$, and an edge between x_1 and x_2 whenever F contains a clause involving both x_1 and x_2). Next we check whether x can be shown to be dependent on the remaining variables in G: this could be either a purely syntactic check or involve a SAT invokation. When x is indeed dependent, then x is added to Z, and moreover all the clauses involved into showing this dependency are removed from F (for simplicity in the algorithm we remove all clauses of G, but a more refined analysis is also possible). This step is important to avoid cyclic dependencies.

Proposition 4. *Let Z be an outcome of Algorithm 1. Then $X \setminus Z$ is an independent support for F. Moreover, let S be a minimal independent support of F relative to the over-approximation $X \setminus Z$. Then S is also a minimal independent support of F.*

The first part of Proposition 4 summarizes the correctness of Algorithm 1. The second part shows that the output of the algorithm can be used to obtain an over-approximation of a minimal independent support – and thus it can be viewed as a preprocessing step for computing minimal independent support.

4.4 Combined Algorithm

Algorithm MIS (Algorithm 2) presents our combined approach to compute a minimal independent support. The algorithm accepts a formula F in CNF, and both an under-approximation U and an over-approximation V. We require that $U \subseteq V$ and that V is an independent support for F. As the first step, we call FindLocalDependencies described in Section 4.3 to compute a set of (locally) dependent variables, which is essentially used to further refine the over-approximation V. Next, following the description in Section 4.2, we translate the problem into a GMUS computation. The call to ComputeGMUS refers to a state-of-the-art algorithm to compute GMUSes (in our experiments, we use MUSer2). The independent support returned by the algorithm consists of the variables in the under-approximation U and the variables that correspond to the groups in the minimal group-unsatisfiable subset. The correctness of this algorithm follows from Proposition 3.

Given the computationally expensive nature of GMUS computation, it may happen that ComputeGMUS exceeds a specified time-limit. However, it is important to note that MUSer2 still returns a sound over-approximation of a minimal group-unsatisfiable subset in case of a time-out (as it employs a variant of the *deletion-based* approach described in [25]). In this case the support consisting

Algorithm 2 MIS(F, U, V)

```
Input: CNF formula $F$; sets $U$,$V$ such that $U \subseteq V \subseteq \mathsf{Vars}(F)$ and $V$ is independent support for $F$
Output: Minimal $S$ with the property that $U \subseteq S \subseteq V$ and $S$ is an independent support for $F$
1: Z = $\mathsf{FindLocalDependencies}(F, V)$
2: $\{\Omega, H_1, \dots, H_n\} = \mathsf{TranslateToGMUS}(F, U, V \setminus Z)$
3: $\{H_{i_1}, \dots, H_{i_n}\} = \mathsf{ComputeGMUS}(\{\Omega, H_1, \dots, H_n\})$
4: $S = U \cup \{x_{i_1}, \dots, x_{i_n}\}$
5: return $S$
```

of the variables in U and the variables in the computed over-approximation returned by ComputeGMUS is still an independent support. Therefore, MIS behaves as an *anytime* algorithm; that is, it always returns a sound independent support for a given time budget. Our experiments indicate that this anytime behavior is useful in computing independent supports – even if these are not minimal, they are significantly smaller than the support of F and improve performance of sampling and counting tools by 2-3 orders of magnitude.

4.5 Computation of Minimum Independent Support

Since the problem of computing a minimum-sized independent support can be reduced to that of computing minimum-sized group-unsatisfiable subset, we can extend MIS to compute a minimum-sized independent support, by following the two modifications below. First, we remove the call to FindLocalDependencies – as this is a greedy heuristic that provide guarantees of minimality but not of mimum size. Second, we replace the call to compute minimal group-unsatisfiable subset with the call to compute minimum group-unsatisfiable subset. We use SMIS to denote the resulting algorithm. Our experimental comparison of MIS and SMIS, discussed in Section 5, shows that MIS scales to larger formulas, while SMIS computes even smaller sized independent supports for a subset of benchmarks that are within its reach.

4.6 Handling User Input

In some of our applications the user is allowed to additionally provide a set of variables W that is believed to form an independent support of F, and the task is to minimize this set. There are two interesting scenarios associated with this. If W is indeed an independent support of F, as can be checked by checking satisfiability of $Q_{F,W}$, then W can be used an as over-approximation of an independent support, that is, one can look for a minimal independent support relative to the over-approximation prescribed by W. It is possible, however, that W is not really an over-approximation. In our experience, in these cases the user input is still "close" to being correct, and so we suggest the following two-step approach. First, we treat W as an under-approximation and find a minimal set

U such that $W \cup U$ forms an independent support. Second, we treat $W \cup U$ as an over-approximation and find a minimal subset of $W \cup U$. In our experience, not only does this scheme results in a minimal independent support that is close to the user input, but is also significantly faster than computing a minimal independent support from scratch

4.7 Applications to Sampling and Counting

Chakraborty et al. [9] showed that H_{xor} constructed over an independent support I, denoted H_{xor}^{I}, is 3-universal. Therefore, we can replace the hash functions employed in recent 3-universal hashing-based approaches to sampling such as UniGen [9], UniGen2 [10], PAWS [13] with H_{xor}^{I} with *no loss* of theoretical guarantees. Similarly, hashing-based counting techniques can also be augmented with H_{xor}^{I}. In the next section, we compare the performance of UniGen2 vis-a-vis IUniGen2 and ApproxMC vis-a-vis IApproxMC, where the IUniGen2 and IApproxMC are UniGen2 and ApproxMC augmented with H_{xor}^{I} respectively.

Since counting techniques require weaker guarantees of universality, recent research efforts have focused on obtaining precise bounds on the size of XORs required. These efforts are motivated by investigations into shorter XORs by Gomes et al. [16]. It was empirically demonstrated that short XORs, surprisingly, perform quite well for wide variety of benchmarks. The earlier works [16,12] failed, however, to obtain provable bounds on adequate size of XOR constraints that are close to empirical observations. In contrast, our results imply that we can substitute n by $|\mathsf{I}|$ in Theorem 3 of [12] to obtain new provable bounds on the size of XORs required for approximate model counting.

5 Evaluation

To evaluate the performance and impact of MIS, we built a prototype implementation[5] in C++ and conducted an extensive set of experiments on diverse set of public-domain problem instances. In these experiments, a typical instance is a formula F, with set of support X, and independent support I computed by MIS. The main objectives of our experimental set up was to seek answers for the following questions:

1. How do MIS and SMIS scale to large formulas and how do sizes of I computed by MIS and SMIS compare to X?
2. How does the performance and size of computed I vary with the user input?
3. How does employing H_{xor} on I instead of X affect the performance of UniGen2 and ApproxMC, the state-of-the-art sampling and counting tools?
4. How do new provable bounds on the size of XORs required for approximate model counting techniques compare with previously known bounds?

[5] The tool along with source code is available at `http://www.cs.rice.edu/CS/Verification/Projects/CUSP/`

In summary, we observe that MIS scales to large formulas with tens of thousands of variables, and the minimal independent support computed by MIS are typically of 1/10 to 1/100 the size of support of the formulas. Furthermore, utilizing user input even when the initial user input is only an under-approximation, MIS can compute minimal independent supports significantly faster than without user input. Employing 3-universal hash functions H_{xor} over I resulted in 2-3 orders of magnitude performance improvement of UniGen2 and ApproxMC. Finally, by utilizing I computed by MIS and SMIS, we provide the first theoretically proven bounds on size of XOR constraints that are close to empirically observed bounds.

5.1 Experimental Setup

We conducted experiments on a heterogeneous suite of benchmarks used in earlier works on sampling and counting [9]. The benchmark suite employed in the experiments consisted of problems arising from probabilistic inference in grid networks, synthetic grid-structured random interaction Ising models, plan recognition, DQMR networks, bit-blasted versions of SMTLIB benchmarks, ISCAS89 combinational circuits with weighted inputs, and program synthesis examples. We employed MUSer2 [4] for group minimal group-unsatisfiable subset computation and forqes [1] for group minimum-unsatisfiable subset computation. We used a high-performance cluster to conduct multiple experiments in parallel. Each node of the cluster had a 12-core 2.83 GHz Intel Xeon processor, with 4GB of main memory, and each of our experiments was run on a single core. Since different runs of MIS compute different minimal independent supports depending on the input from pseudo-random generator, we compute up to five independent supports for each benchmark and report the median of corresponding statistics.

5.2 Results

Runtime Performance of MIS and SMIS

Table 1 presents the runtime of MIS and SMIS for a subset of the benchmarks. The names of the benchmarks are specified in column 1, while columns 2 and 3 list the number of variables and clauses for each benchmark. Column 4 and 6 list the median runtime and median size of minimal independent supports (I) computed by MIS. Column 6 lists the ratio of the number of variables to $|\mathsf{I}|$. Column 7 and 8 list the runtime and size of a minimum-sized independent support (I_m). The ratio of $|\mathsf{I}_m|$ to $|\mathsf{I}|$ is presented in column 9. The results demonstrate that MIS scales to fairly large formulas, and the minimal independent supports computed by MIS are one to two orders of magnitude compared to the overall support. The comparison of MIS vis-a-vis SMIS highlights a tradeoff in performance. In particular, while MIS scales to larger formulas, SMIS computes even smaller independent supports for a subset of benchmarks that are within its reach (and in some cases removes up to 40% additional variables).

Benchmark	**#vars**	**#clas**	MIS **time(s)**	$\lvert\mathsf{I}\rvert$	$\frac{\#vars}{\lvert\mathsf{I}\rvert}$	SMIS **time(s)**	$\lvert\mathsf{I}_m\rvert$	$\frac{\lvert\mathsf{I}_m\rvert}{\lvert\mathsf{I}\rvert}$
squaring4	891	2839	868.71	55	16.05	1174.46	36	0.65
s953a_15_7	602	1657	7.48	48	12.41	11.03	45	0.93
squaring30	1031	3693	192.14	30	34.37	144.82	29	0.97
case_2_b12_1	427	1385	1.42	34	12.56	16.52	30	0.88
scenarios_llreverse	1096	4217	59.8	81	13.45	205.0	46	0.56
squaring10	1099	3632	3321.29	56	19.45	1609.63	40	0.71
TR_ptb_1_linear	1969	6288	1297.77	122	16.07	768.37	106	0.87
s1488_7_4	872	2499	11.38	24	36.33	–	–	–
s5378a_15_7	3766	8732	1990.1	227	16.59	–	–	–
lssBig	12438	149909	536.88	46	270.39	–	–	–
blockmap_10_02.net	12562	26022	2637.74	78	161.05	–	–	–
lss	13373	156208	971.24	45	297.18	–	–	–
blockmap_10_03.net	13786	28826	13442.28	125	110.29	–	–	–
20	13887	60046	40.29	51	272.29	14.6	50	0.98
scenarios_tree_insert_search	16573	61922	18000	943	17.57	–	–	–
blockmap_15_01.net	33035	67424	781.94	49	674.18	–	–	–
blockmap_20_01.net	78650	160055	2513.32	67	1173.88	–	–	–

Table 1. Runtime performance of MIS and SMIS

Impact of User Input on MIS

To study the impact of user input on MIS, we experimented with the suite of benchmarks for which independent support was provided by the sources. Table 2 presents the result of our experiments. Column 1 lists the benchmark while columns 2 and 3 list the number of variables and clauses for each benchmark. Columns 4 and 5 list the runtime and the median size of computed I by MIS without user input. Columns 6–9 report statistics when the user input is provided to MIS. Column 6 lists the size of I provided by the user while column 7 and 8 present the runtime and the size of computed I by MIS. Column 9 lists the fraction of ratio of intersection of computed I and user-provided I to the computed I. We use “U” and ”O” to denote that the input provided by user was an under-approximation and over-approximation of an independent support respectively.

Table 2 shows that user-provided input are not necessarily minimal and are sometimes under approximation of an independent support. Since several minimal independent supports exist, it does not necessarily imply that size of an under-approximation would be smaller than every minimal independent support; e.g., for benchmark “Pollard”, while oen of the independent supports is of size 48, the inpt with size 50 is not an independent support and is, therefore, an under-approximation of some other independent support. Table 2 clearly demonstrates that MIS is able to take advantage of user input, even when the initial user input is only an under approximation, and can compute I significantly faster than without user input. Since initial user input is only an under approximation in several cases and therefore, algorithmic techniques such as MIS are required to compute a sound independent support.

			Without User Input		With User Input			
Benchmark	**#vars**	**#clas**	**MIS time(s)**	**|I|**	**User |I|**	**MIS time(s)**	**Computed |I|**	**Type**
TR_b14_2_linear	1570	4963	243.65	136	204	234.0	103	O
squaring7	1628	5837	12329.2	58	72	4404.22	40	O
55	1874	8384	0.1	38	46	0.24	38	U
TR_b12_1_linear	1914	6619	5963.92	73	99	1559.43	60	U
TR_b12_2_linear	2426	8373	15505.02	79	107	1779.25	64	O
TR_device_1_even_linear	2447	7612	612.19	176	281	338.06	158	O
case_1_b12_even1	2681	8492	4507.71	155	150	1534.94	147	O
case_2_b12_even1	2681	8492	4249.56	149	150	2008.88	147	O
scenarios_tree_insert_insert	2797	10427	837.14	101	84	725.08	85	U
Pollard	2800	49543	1211.4	179	50	543.94	48	U
56	2801	9965	2.23	37	38	1.84	37	U
ProcessBean	3130	11689	172.64	305	166	92.44	156	U
scenarios_tree_delete2	3411	12783	444.61	179	138	389.79	137	U
lss_harder	3465	62713	1727.77	116	21	1690.61	22	U
s5378a_15_7	3766	8732	1990.1	227	214	559.06	214	O
listReverseEasy	4092	15867	16715.34	144	121	1959.57	99	U
reverse	9485	535676	25.03	201	262	24.2	195	U
lss	13373	156208	971.24	45	20	665.22	20	U
110	15316	60974	9.2	80	88	9.08	80	U

Table 2. Impact of User Input on MIS. "U" and "O" denote that the input provided by user was an under-approximation and over-approximation of an independent support respectively.

Impact on Performance of Sampling and Counting Techniques

We compared the performance of UniGen2 with IUniGen2 and of ApproxMC with IApproxMC. We used an overall timeout of 5 hours, and the tolerance (ε) for UniGen2 and IUniGen2 was set to 16, while tolerance (ε) and confidence ($1 - \delta$) were set to 0.8 and 0.8, respectively, for ApproxMC and IApproxMC. The parameter values were chosen to match the corresponding values in previously published works on ApproxMC [8] and UniGen2 [10]. In summary, while either ApproxMC or UniGen2 timed out on 36 out of 112 benchmarks, either IApproxMC or IUniGen2 were able to count or sample respectively on all the benchmarks. Since we computed up to five independent supports for each benchmark, we also computed range of runtime for IApproxMC and IUniGen2.

Table 3 presents the comparison of runtimes of UniGen2 and IUniGen2 as well as ApproxMC and IApproxMC for a subset of the benchmarksColumn 1 lists the benchmarks, while column 2 report the number of variables for each benchmark. Column 3 lists the runtime of MIS to compute I. Column 4 lists the runtime of ApproxMC, while the median runtime and range of runtimes for IApproxMC are listed in columns 5 and 6. while column 7 lists the runtime of UniGen2 and columns 8 and 9 lists the range of runtimes for IUniGen2. (We generated 100 samples for each benchmark, and sampling time is amortized per sample.) We use '–' to denote the timeout (5 hours).

Table 3 clearly demonstrates that employing 3-universal hash functions H_{xor} over I resulted in 2-3 orders of magnitude performance improvement for both counting and sampling. It is worth noting that for the case of "squaring14", MIS times out, but the over-approximation returned by MIS still allows IUniGen2 and

		MIS	ApproxMC	IApproxMC		UniGen2	IUniGen2	
Benchmark	**#vars**	**time(s)**	**time (s)**	**Median time (s)**	**Range time(s)**	**time (s)**	**Median time (s)**	**Range time(s)**
squaring4	891	868.71	–	1550.04	986.47	–	0.58	0.78
s953a_15_7	602	7.48	–	1221.22	250.72	239.85	0.71	0.33
squaring30	1031	192.14	29974.19	89.82	42.23	11.59	0.24	0.03
case_2_b12_1	427	1.42	1449.15	212.82	82.42	2.56	0.24	0.05
squaring10	1099	3321.29	–	3135.01	3800.54	–	0.83	0.47
s1196a_7_4	708	35.44	–	314.21	167.29	245.21	0.37	0.13
s1238a_7_4	704	47.59	–	404.54	93.32	1036.61	0.4	0.11
case_0_b12_2	827	34.87	–	1528.17	4418.18	–	0.56	0.08
case_1_b12_2	827	23.87	–	1541.06	399.75	–	0.73	0.11
scenarios_llreverse	1096	59.8	–	17109.1	10040.38	–	24.77	360.4
case_2_b12_2	827	25.32	–	1228.28	872.1	–	0.59	0.17
lss_harder	3465	1727.77	13116.78	120.46	301.58	104.91	2.0	2.65
BN_57	1154	103.22	–	517.27	1118.89	200.07	0.32	0.21
BN_59	1112	104.85	–	484.79	236.62	404.29	0.34	0.09
BN_65	925	29.64	–	1322.17	261.33	4220.31	0.7	174.97
squaring1	891	718.78	–	1480.99	296.37	–	0.73	0.34
squaring8	1101	3453.48	–	2061.31	4970.9	–	0.71	0.36

Table 3. Runtime comparison of UniGen2 vis-a-vis IUniGen2 and ApproxMC vis-a-vis IApproxMC

IApproxMC to sample and count, while UniGen2 and ApproxMC timed out. Furthermore, the considerably smaller range of runtimes for most of the benchmarks illustrate the dominating effect of minimal independent supports on the runtime performance. This observation is, however, not always true and we observe that there are cases where the range is considerably large – a detailed analysis is beyond the scope of this work and is left for future work.

Quality of Distribution To measure the impact of H_{xor}^{I} on the quality of distribution generated by UniGen2, we compared the distributions generated by UniGen2 and IUniGen2. We generated a large number N ($\geq 5.6 \times 10^6$) of samples for each benchmark using both UniGen2 and IUniGen2. Since we chose N much larger than $|R_F|$, all witnesses occurred multiple times in the list of samples. We then computed the frequency of generation of individual witnesses, and counted the number if distinct witnesses for each frequency. Plotting the distribution of frequencies — that is, plotting points (x, y) to indicate that each of x distinct witnesses were generated y times — gives a convenient way to visualize the distribution of the samples. Figure 1 depicts this for one representative benchmark (case110, with 16,384 solutions).

It is clear from Figure 1 that the distribution generated by IUniGen2 is practically indistinguishable from that of UniGen2. For the example shown in Fig. 1, the Jensen-Shannon distance between the distributions from IUniGen2 and UniGen2 is 0.0035. These small Jensen-Shannon distances make the distribution of IUniGen2 statistically indistinguishable from that of UniGen2 (see Section IV(C) of [17]).

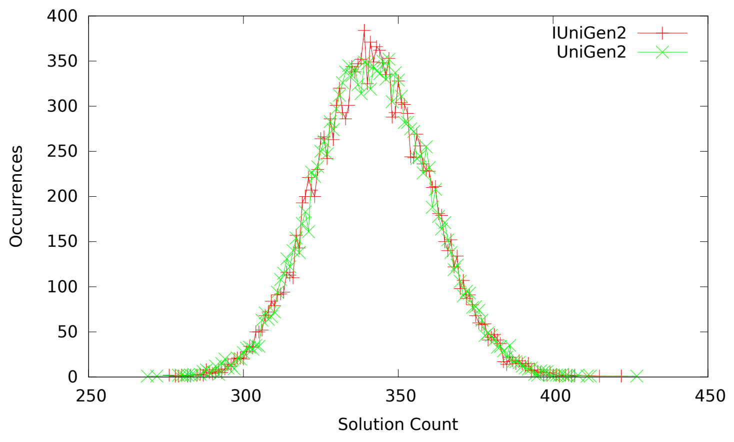

Fig. 1. Uniformity comparison of IUniGen2 and UniGen2

Impact on XOR Size Bounds for Model Counting Techniques

Since approximation techniques for model counting only requires weaker guarantees of universality [8], several techniques have been proposed on employing shorter XORs for model counting [16,12]. The investigations into shorter XORs [16,12] empirically demonstrated that short XORs, surprisingly, perform quite well for wide variety of benchmarks, even without a theoretical guarantee, but have failed to obtain provable bounds on adequate size of XOR constraints that are close to empirical observations. By computing the size of XOR constraints based on the size of minimal independent support and then applying Theorem 3 of [12], we provide the first theoretically proven bounds on adequate size of XOR constraints that are very close to empirically observed bounds.

Table 4 presents the comparison of new theoretical bounds with previously known best theoretical and empirical bounds for benchmarks reported in previous works [16,12]. Column 1 lists the benchmarks, while column 2 and 3 report the number of variables and clauses for each benchmark. Column 4 and 5 present previously known theoretical and empirical bounds on size of XORs [12]. Finally, the new theoretical bounds based on computation of independent supports is presented in column 6. Table 4 clearly shows that new bounds obtained based on minimal independent supports computed by MIS greatly improve on the previously reported theoretical bounds. Furthermore, the bounds are very close to empirically observed bounds. In fact, in one case we obtain theoretical bound that is better than the best known empirical bounds. (It is worth noting that previous results [16,12] on shorter XORs do not extend to sampling techniques as sampling requires stronger guarantees of universality.)

			Previous Bounds		New Bounds
Benchmark	**#vars**	**#clas**	**Theoretical**	**Empirical**	**Theoretical**
ls7R34med	119	622	46	3	12
ls7R35med	136	745	53	3	16
ls7R36med	149	870	56	3	18
log.c.red	352	1933	112	28	9
2bitmax_6	252	766	26	8	21
blk-50-3-10-20	50	30	10	5	5
blk-50-10-3-20	50	30	8	3	5

Table 4. Comparison of bounds on shorter XORs for model counting

6 Conclusion

The recent surge of interest in hashing-based approximate techniques have highlighted the construction of efficient hash functions as a key challenging problem. As a result, hash functions constructed over an independent support of a formula hold promise. In this paper, we present the first algorithmic procedure and corresponding tool, MIS, to determine minimal independent support via reduction to Group MUS. The experimental evaluation over an extensive suite of benchmarks demonstrate that MIS scales to large formulas. Furthermore, the minimal independent supports computed by MIS lead to 2-3 orders of magnitude improvement in the performance of the state-of-the-art sampling tool, UniGen2 and the state-of-the-art model counting tool, ApproxMC. Finally, construction of XORs over independent support allows us to obtain tight theoretical bounds on the size of XOR constraints for approximate model counting – in some cases, even better than previously observed empirical bounds.

7 Acknowledgements

The authors would like to thank anonymous reviewers for their detailed, insightful comments and suggestions that helped improve this manuscript significantly. Sharad Malik's work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. Kuldeep S. Meel and Moshe Y. Vardi's work was supported in part by NSF grants CNS 1049862, CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097.

References

1. M. L. J. M.-S. A. Ignatiev, A. Previti. Smallest mus extraction with minimal hitting set dualization. In *Proc. of CP (To Appear)*, 2015.
2. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. of FOCS*, pages 340–351, 2003.

3. A. Belov and J. Marques-Silva. Generalizing redundancy in propositional logic: Foundations and hitting sets duality. *CoRR*, abs/1207.1257, 2012.
4. A. Belov and J. Marques-Silva. Muser2: An efficient MUS extractor. *JSAT*, 8(3/4):123–128, 2012.
5. J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proc. of ACM symposium on Theory of computing*, pages 106–112. ACM, 1977.
6. S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Proc. of AAAI*, pages 1722–1730, 2014.
7. S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. In *Proc. of CAV*, volume 8044, pages 608–623, 2013.
8. S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, volume 8124, pages 200–216. Springer, 2013.
9. S. Chakraborty, K. S. Meel, and M. Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, pages 1–6, 2014.
10. S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. On parallel scalable uniform sat witness generation. In *Proc. of TACAS*, pages 304–319, 2015.
11. C. Domshlak and J. Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30(1):565–620, 2007.
12. S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, pages 271–279, 2014.
13. S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Embed and project: Discrete sampling with universal hashing. In *Proc. of NIPS*, pages 2085–2093, 2013.
14. S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*, pages 334–342, 2013.
15. Z. Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *Prof. of VLSID*, pages 37–42, 2007.
16. C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. Short xors for model counting: from theory to practice. In *Proc. of SAT*, pages 100–106, 2007.
17. I. Grosse, P. Bernaola-Galván, P. Carpena, R. Román-Roldán, J. Oliver, and H. E. Stanley. Analysis of symbolic sequences using the Jensen-Shannon divergence. *Physical Review E*, 65(4):041905, 2002.
18. M. Jerrum and A. Sinclair. The Markov Chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems*, pages 482–520, 1996.
19. M. Jerrum, L. Valiant, and V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2-3):169–188, 1986.
20. N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *Proc. of ICCAD*, pages 258–265, 2007.
21. P. Liberatore. Redundancy in logic I: CNF propositional formulae. *Artif. Intell.*, 163(2):203–232, 2005.
22. M. H. Liffiton, M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. Marques-Silva, and K. A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints*, 14(4):415–442, 2009.
23. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

24. D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Md., 1983.
25. J. Marques-Silva. Computing minimally unsatisfiable subformulas: State of the art and future directions. *Multiple-Valued Logic and Soft Computing*, 19(1-3):163–183, 2012.
26. A. Nadel. Boosting minimal unsatisfiable core extraction. In *Proc. 10th Int. Conf. on Formal Methods in Computer-Aided Design*, pages 221–229, 2010.
27. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. Constraint-based random stimuli generation for hardware verification. In *Proc of IAAI*, pages 1720–1727, 2006.
28. C. P. Gomes, A. Sabharwal, and B. Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Proc. of NIPS*, pages 670–676, 2007.
29. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3dd Ed.).* Prentice Hall, 2009.
30. T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT*, 2004.
31. T. Sang, P. Bearne, and H. Kautz. Performing bayesian inference by weighted model counting. In *Prof. of AAAI*, pages 475–481, 2005.
32. L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
33. Y. Xue, A. Choi, and A. Darwiche. Basing decisions on sentences in decision diagrams. In *Proc. of AAAI*, 2012.