INC: A Scalable Incremental Weighted Sampler

Suwei Yang
National University of Singapore
Singapore
suwei.yang@comp.nus.edu.sg

Victor Liang
GrabTaxi Holdings Pte. Ltd.
Singapore
victor.liang@grab.com

Kuldeep S. Meel
National University of Singapore
Singapore
meel@comp.nus.edu.sg

***Abstract*—The fundamental problem of weighted sampling involves sampling of satisfying assignments of Boolean formulas, which specify sampling sets, and according to distributions defined by pre-specified weight functions to weight functions. The tight integration of sampling routines in various applications has highlighted the need for samplers to be incremental, i.e., samplers are expected to handle updates to weight functions.**

The primary contribution of this work is an efficient knowledge compilation-based weighted sampler, INC[1], designed for incremental sampling. INC builds on top of the recently proposed knowledge compilation language, OBDD[∧], and is accompanied by rigorous theoretical guarantees. Our extensive experiments demonstrate that INC is faster than state-of-the-art approach for majority of the evaluation. In particular, we observed a median of $1.69\times$ runtime improvement over the prior state-of-the-art approach.

***Index Terms*—knowledge compilation, sampling, weighted sampling**

I. Introduction

Given a Boolean formula F and weight function W, weighted sampling involves sampling from the set of satisfying assignments of F according to the distribution defined by W. Weighted sampling is a fundamental problem in many fields such as computer science, mathematics and physics, with numerous applications. In particular, constrained-random simulation forms the bedrock of modern hardware and software verification efforts [1].

Sampling techniques are fundamental building blocks, and there has been sustained interest in the development of sampling tools and techniques. Recent years witnessed the introduction of numerous sampling tools and techniques, from approximate sampling techniques to uniform samplers SPUR and KUS, and weighted sampler WAPS [2]–[6]. Sampling tools and techniques have seen continuous adoption in many applications and settings [7]–[12]. The scalability of a sampler is a consideration that directly affects its adoption rate. Therefore, improving scalability continues to be a key objective for the community focused on developing samplers.

The tight integration of sampling routines in various applications has highlighted the importance for samplers to handle incremental weight updates over multiple sampling rounds, also known as incremental weighted sampling. Existing efforts on improving scalability typically focus on single round weighted sampling, and might have overlooked the incremental setting. In particular, existing approaches involving incremental weighted sampling typically employ off-the-shelf weighted samplers which could lead to less than ideal incremental sampling performance.

The primary contribution of this work is an efficient scalable weighted sampler INC that is designed from the ground up to address scalability issues in incremental weighted sampling settings. The core architecture of INC is based on knowledge compilation (KC) paradigm, which seeks to succinctly represent all satisfying assignments of a Boolean formula with a directed acyclic graph (DAG) [13]. In the design of INC, we make two core decisions that are responsible for outperforming the current state-of-the-art weighted sampler. Firstly, we build INC on top of PROB (Probabilistic OBDD[∧] [14]) which is substantially smaller than the KC diagram used in the prior state-of-the-art approaches. Secondly, INC is designed to perform *annotation*, which refers to the computation of joint probabilities, in log-space to avoid the slower alternative of using arbitrary precision math computations.

Given a Boolean formula F and weight function W, INC compiles and stores the compiled PROB in the first round of sampling. The weight updates for subsequent incremental sampling rounds are processed without recompilation, amortizing the compilation cost. Furthermore, for each sampling round, INC simultaneously performs *annotation* and sampling in a single bottom-up pass of the PROB, achieving speedup over existing approaches. We observed that INC is significantly faster than the existing state-of-the-art in the incremental sampling routine. In our empirical evaluations, INC achieved a median of $1.69\times$ runtime improvement over the state-of-the-art weighted sampler, WAPS [6]. Additional performance breakdown analysis supports our design choices in the development of INC. In particular, PROB is on median $4.64\times$ smaller than the KC diagram used by the competing approach, and log-space *annotation* computations are on median $1.12\times$ faster than arbitrary precision computations. Furthermore, INC demonstrated significantly better handling of incremental sampling rounds, with incremental sampling rounds to be on median 5.9% of the initial round, compared to 67.6% for WAPS.

The rest of the paper is organized as follows. We first introduce the relevant background knowledge and related works in Section II. We then introduce PROB and its properties in Section III. In Section IV, we introduce our weighted sampler INC, detail important implementation decisions, and provide theoretical analysis of INC. We then describe the extensive

[1]code available at https://github.com/grab/inc-weighted-sampler/

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_27

empirical evaluations and discuss the results in Section V. Finally, we conclude in Section VI.

II. Background and Related Work

Knowledge Compilation: Knowledge compilation (KC) involves representing logical formulas as directed acyclic graphs (DAG), which are commonly referred to as knowledge compilation diagrams [13]. The goal of knowledge compilation is to allow for tractable computation of certain queries such as model counting and weighted sampling. There are many well-studied forms of knowledge compilation diagrams such as d-DNNF, SDD, BDD, ZDD, OBDD, AOBDD, and the likes [15]–[21]. In this work, we build our weighted sampler upon a variant of OBDD known as OBDD[$\wedge$] [14].

OBDD[$\wedge$]: Lee [15] introduced Binary Decision Diagram (BDD) as a way to represent Shannon expansion [22]. [16] introduced fixed variable orderings to BDDs (known as OBDD) [16] for canonical representation and compression of BDDs via shared sub-graphs. Lai et al. [14] introduced conjunction nodes to OBDDs (known as OBDD[$\wedge$]) [14] to further reduce the size of the resultant DAG to represent a given Boolean formula. In this work, we parameterize an OBDD[$\wedge$] to form a PROB that is used for weighted sampling.

Sampling: A Boolean variable x can be assigned either *true* or *false*, and its literal refers to either x or its negation. A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, with each clause being a disjunction of literals. A Boolean formula F is satisfiable if there exists an assignment τ of its variables such that the F evaluates to *true*. The model count of Boolean formula F refers to the number of distinct satisfying assignments of F.

Weighed sampling concerns with sampling elements from a distribution according to non-negative weights provided by a user-defined weight function W. In the context of this work, weighted sampling refers to the process of sampling from the space of satisfying assignments of a Boolean formula F. The weight function W assigns a non-negative weight to each literal l of F. The weight of an assignment τ is defined as the product of the weight of its literals.

WAPS: KUS [5] utilizes knowledge compilation techniques, specifically Deterministic Decomposable Negation Normal Form (d-DNNF) [19], to perform uniform sampling in 2 passes of the d-DNNF. *Annotation* is performed in the first pass, followed by sampling. WAPS [6] improves upon KUS by enabling weighted sampling via parameterization of the d-DNNF. WAPS performs sampling in a similar manner to KUS, the main difference being that the *annotation* step in WAPS takes into account the provided weight function. In contrast, we introduce INC which performs weighted sampling in a single pass by leveraging the DAG structure of PROB.

Knowledge compilation-based samplers typically perform incremental sampling as follows. The sampling space is first expressed as satisfying assignments of a Boolean formula, which is then compiled into the respective knowledge compilation form. In the following step, samples are drawn according to the given weight function W. Subsequently, the weights are updated depending on application logic and weighted sampling is performed again. The process is repeated until an application-specific stopping criterion is met. An example of such an application would be the Baital framework [10], developed to use incremental weighted sampling to generate test cases for configurable systems.

III. PROB: - Probabilistic OBDD[$\wedge$]

PROB is a DAG composed of four types of nodes - *conjunction*, *decision*, *true* and *false* nodes. The internal nodes of a PROB consist of conjunction and decision nodes whereas the leaf nodes of the PROB consist of true and false nodes. A PROB is recursively made up of sub-PROBs that represent sub-formulas of Boolean formula F. We use $\mathsf{VarSet}(n)$ to refer to the set of variables of F represented by a PROB with n as the root node. $\mathsf{Subdiagram}(n)$ refers to the sub-PROB starting at node n and $\mathsf{Parent}(n)$ refers to the immediate parent of node n in PROB.

A. PROB *Structure*

Conjunction node ($\wedge$-node): A $\wedge$-node n_c represents conjunctions in the assignment space. There are no limits to the number of child nodes that n_c can have. However, the set of variables ($\mathsf{VarSet}(\cdot)$) of each child node of n_c must be disjoint. An example of a $\wedge$-node would be $n2$ in Figure 1. Notice that $\mathsf{VarSet}(n4) = \{z\}$ and $\mathsf{VarSet}(n5) = \{y\}$ are disjoint.

Decision node: A decision node n_d represents decisions on the associated Boolean variable $\mathsf{Var}(n_d)$ in Boolean formula F that the PROB represents. A decision node can have exactly two children - *lo-child* ($\mathsf{Lo}(n_d)$) and *hi-child* ($\mathsf{Hi}(n_d)$). $\mathsf{Lo}(n_d)$ represents the assignment space when $\mathsf{Var}(n_d)$ is set to *false* and $\mathsf{Hi}(n_d)$ represents otherwise. $\theta_{n_{d_{hi}}}$ and $\theta_{n_{d_{lo}}}$ refer to the parameters associated with the edge connecting decision node n_d with $\mathsf{Hi}(n_d)$ and $\mathsf{Lo}(n_d)$ respectively in a PROB. Node $n1$ in Figure 1 is a decision node with $\mathsf{Var}(n1) = x$, $\mathsf{Hi}(n1) = n3$ and $\mathsf{Lo}(n1) = n2$.

True and False nodes: True ($\top$) and false ($\bot$) nodes are leaf nodes in a PROB. Let τ be an assignment of all variables of Boolean formula F and let PROB ψ represent F. τ corresponds to a traversal of ψ from the root node to leaf nodes. The traversal follows τ at every decision node and visits all child nodes of every conjunction node encountered along the way. τ is a satisfying assignment if all parts of the traversal eventually lead to the true node. τ is not a satisfying assignment if any part of the traversal leads to the false node. With reference to Figure 1, let $\tau_1 = \{x, y, \neg z\}$ and $\tau_2 = \{x, y, z\}$. For τ_1, the traversal would visit $n1, n3, n6, n7, n9$, and τ_1 is a satisfying assignment since the traversal always leads to $\top$ node ($n9$). As a counter-example, τ_2 is not a satisfying assignment with its corresponding traversal visiting $n1, n3, n6, n7, n8, n9$. τ_2 traversal visits $\bot$ node ($n8$) because variable $z \mapsto$ *true* in τ_2 and $\mathsf{Hi}(n6)$ is node $n8$.

B. PROB *Parameters*

In the PROB structure, each decision node n_d has two parameters $\theta_{\mathsf{Lo}(n_d)}$ and $\theta_{\mathsf{Hi}(n_d)}$, associated with the two branches

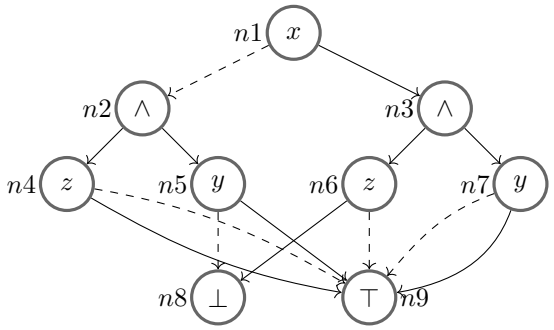


Fig. 1: A smooth PROB ψ_1 with 9 nodes, $n1, ..., n9$, representing $F = (x \vee y) \wedge (\neg x \vee \neg z)$. Branch parameters are omitted

of n_d, which sums up to 1. $\theta_{\mathsf{Lo}(n_d)}$ is the normalized weight of the literal $\neg\mathsf{Var}(n_d)$ and similarly, $\theta_{\mathsf{Hi}(n_d)}$ is that of the literal $\mathsf{Var}(n_d)$. One can view $\theta_{\mathsf{Lo}(n_d)}$ to be the probability of picking $\neg\mathsf{Var}(n_d)$ and $\theta_{\mathsf{Hi}(n_d)}$ to be that of picking $\mathsf{Var}(n_d)$ by the *determinism* property introduced later. Let x_i be $\mathsf{Var}(n_d)$. Given a weight function W:

$$\theta_{\mathsf{Lo}(n_d)} = \frac{W(\neg x_i)}{W(\neg x_i) + W(x_i)} \quad \theta_{\mathsf{Hi}(n_d)} = \frac{W(x_i)}{W(\neg x_i) + W(x_i)}$$

C. PROB *Properties*

The PROB structure has important properties such as *determinism* and *decomposability*. In addition to the *determinism* and *decomposability* properties, we ensure that PROBs used in this work have the *smoothness* property through a smoothing process (Algorithm 1).

Property 1 (Determinism)**.** *For every decision node n_d, the set of satisfying assignments represented by* $\mathsf{Hi}(n_d)$ *and* $\mathsf{Lo}(n_d)$ *are logically disjoint.*

Property 2 (Decomposability)**.** *For every conjunction node n_c,* $\mathsf{VarSet}(c_i) \cap \mathsf{VarSet}(c_j) = \emptyset$ *for all c_i and c_j where $c_i, c_j \in \mathsf{Child}(n_c)$ and $c_i \neq c_j$.*

Property 3 (Smoothness)**.** *For every decision node n_d,* $\mathsf{VarSet}(\mathsf{Hi}(n_d)) = \mathsf{VarSet}(\mathsf{Lo}(n_d))$.

D. Joint Probability Calculation with PROB

In Section III-B, we mention that one can view the branch parameters as the probability of choosing between the positive and negative literal of a decision node. Notice that because of the *decomposability* and *determinism* properties of PROB, it is straightforward to calculate the joint probabilities at every given node. At each conjunction node n_c, since the variable sets of the child nodes of n_c are disjoint by *decomposability*, the joint probability of n_c is simply the product of joint probabilities of each child node. At each decision node n_d, there are only two possible outcomes on $\mathsf{Var}(n_d)$ - positive literal $\mathsf{Var}(n_d)$ or negative literal $\neg\mathsf{Var}(n_d)$. By *determinism* property, the joint probability is the sum of the two possible scenarios. Formally, the calculations for joint probabilities P' at each node in PROB are as follows:

$$P' \text{ of } \wedge\text{-node } n_c = \prod_{c \in \mathsf{Child}(n_c)} P'(c) \tag{EQ1}$$

$$P' \text{ of decision-node } n_d = \theta_{\mathsf{Lo}(n_d)} \times P'(\mathsf{Lo}(n_d)) + \theta_{\mathsf{Hi}(n_d)} \times P'(\mathsf{Hi}(n_d)) \tag{EQ2}$$

For true node n, $P'(n) = 1$ because it represents satisfying assignments when reached. In contrast $P'(n) = 0$ when n is a *false* node as it represents non-satisfying assignments. In Proposition 2, we show that weighted sampling is equivalent to sampling according to joint probabilities of satisfying assignments of a PROB.

IV. INC - SAMPLING FROM PROB

In this section, we introduce INC - a bottom-up algorithm for weighted sampling on PROB. We first describe INC for drawing one sample and subsequently describe how to extend INC to draw k samples at once. We also provide proof of correctness that INC is indeed performing weighted sampling. As a side note, samples are drawn with replacement, in line with the existing state-of-the-art weighted sampler [6].

A. *Preprocessing* PROB

In the main sampling algorithm (Algorithm 2) to be introduced later in this section, the input is a smooth PROB. As a preprocessing step, we introduce Smooth algorithm that takes in a PROB ψ and performs smoothing.

The Smooth algorithm processes the nodes in the input PROB ψ in a bottom-up manner while keeping track of $\mathsf{VarSet}(n)$ for every node n in ψ using a map κ. *True* and *false* nodes have $\emptyset$ as they are leaf nodes and do not represent any variables. At each conjunction node, its variable set is the union of variable sets of its child nodes.

The smoothing happens at decision node n in ψ when $\mathsf{VarSet}(\mathsf{Lo}(n))$ and $\mathsf{VarSet}(\mathsf{Hi}(n))$ do not contain the same set of variables as shown by lines 8 and 16 of Algorithm 1. In the smoothing process, a new conjunction node (*lcNode* for $\mathsf{Lo}(n)$ and *rcNode* for $\mathsf{Hi}(n)$) is created to replace the corresponding child of n, with the original child node now set as a child of the conjunction node. Additionally, for each of the missing variables v, a decision node representing v is created and added as a child of the respective conjunction node. The decision nodes created during smoothing have both their lo-child and hi-child set to the *true* node. To reduce memory footprint, we check if there exists the same decision node before creating it in the checkMakeTrueDecisionNode function.

As an example, we refer to ψ_2 in Figure 2. It is obvious that ψ_2 is not smooth, because $\mathsf{VarSet}(\mathsf{Lo}(n1)) = \{y\}$ and $\mathsf{VarSet}(\mathsf{Hi}(n1)) = \{z\}$. In the smoothing process, we replace $\mathsf{Lo}(n1)$ with a new conjunction node $n2$ and add a decision node $n4$ representing missing variable z, with both child set to *true* node $n9$. We repeat the steps for $\mathsf{Hi}(n1)$ to arrive at PROB ψ_1 in Figure 1.

Algorithm 1 Smooth - returns a smoothed PROB

Input: PROB ψ
Output: smooth PROB

1: $\kappa \leftarrow$ initMap()
2: **for** node n of ψ in bottom-up order **do**
3: **if** n is *true* node or *false* node **then**
4: $\kappa[n] \leftarrow \emptyset$
5: **else if** n is $\wedge$-node **then**
6: $\kappa[n] \leftarrow$ unionVarSet(Child$(n), \kappa$)
7: **else**
8: **if** $\kappa[\mathsf{Hi}(n)] - \kappa[\mathsf{Lo}(n)] \neq \emptyset$ **then**
9: lset $\leftarrow \kappa[\mathsf{Hi}(n)] - \kappa[\mathsf{Lo}(n)]$
10: lcNode $\leftarrow new \wedge -node()$
11: lcNode.addChild(Lo(n))
12: **for** var v in lset **do**
13: dNode $\leftarrow$ checkMakeTrueDecisionNode(v)
14: lcNode.addChild(dNode)
15: Lo$(n) \leftarrow$lcNode
16: **if** $\kappa[\mathsf{Lo}(n)] - \kappa[\mathsf{Hi}(n)] \neq \emptyset$ **then**
17: rset $\leftarrow \kappa[\mathsf{Lo}(n)] - \kappa[\mathsf{Hi}(n)]$
18: rcNode $\leftarrow new \wedge -node()$
19: rcNode.addChild(Hi(n))
20: **for** var v in rset **do**
21: dNode $\leftarrow$ checkMakeTrueDecisionNode(v)
22: rcNode.addChild(dNode)
23: Hi$(n) \leftarrow$rcNode
24: $\kappa[n] \leftarrow \mathsf{Var}(n) \cup$ unionVarSet$(\{\mathsf{Hi}(n), \mathsf{Lo}(n)\})$
25: **return** ψ

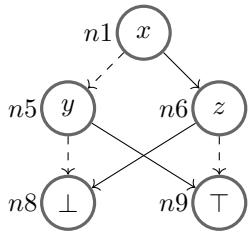


Fig. 2: A PROB ψ_2 representing Boolean formula $F = (x \vee y) \wedge (\neg x \vee \neg z)$, branch parameters are omitted

B. Sampling Algorithm

INC takes a PROB ψ representing Boolean formula F and draws a sample from the space of satisfying assignments of F, the process is illustrated by Algorithm 2. INC performs sampling in a bottom-up manner while integrating the *annotation* process in the same bottom-up pass. Since we want to sample from the space of satisfying assignments we can ignore *false* nodes in ψ entirely by considering a sub-DAG that excludes *false* nodes and edges leading to them, as shown by line 3. As an example, hideFalseNode when applied to ψ_1 would remove node $n8$ and the edges immediately leading to it. Next, INC processes each of the remaining nodes in bottom-up order while keeping two caches - ω to store the partial samples from each node, φ to store the joint probability at each node.

Algorithm 2 INC - returns a satisfying assignment based on PROB ψ parameters

Input: smooth PROB ψ
Output: a sampled satisfying assignment

1: cache $\omega \leftarrow$ initCache()
2: joint prob cache $\varphi \leftarrow$ initCache()
3: $\psi' \leftarrow$ hideFalseNode(ψ)
4: **for** node n of ψ' in bottom-up order **do**
5: **if** n is *true* node **then**
6: $\omega[n] \leftarrow \emptyset$
7: $\varphi[n] \leftarrow 1$
8: **else if** n is $\wedge$-node **then**
9: $\omega[n] \leftarrow$ unionChild(Child$(n), \omega$)
10: $\varphi[n] \leftarrow \prod_{c \in \mathsf{Child}(n)} \varphi[c]$
11: **else**
12: $p_{lo} \leftarrow \theta_{\mathsf{Lo}(n)} \times \varphi[\mathsf{Lo}(n)]$
13: $p_{hi} \leftarrow \theta_{\mathsf{Hi}(n)} \times \varphi[\mathsf{Hi}(n)]$
14: $p_{joint} \leftarrow p_{lo} + p_{hi}$
15: $\varphi[n] \leftarrow p_{joint}$
16: $r \leftarrow x \sim \mathsf{binomial}(1, \frac{p_{hi}}{p_{joint}})$
17: **if** r is 1 **then**
18: $\omega[n] \leftarrow \omega[\mathsf{Hi}(n)] \cup \mathsf{Var}(n)$
19: **else**
20: $\omega[n] \leftarrow \omega[\mathsf{Lo}(n)] \cup \neg\mathsf{Var}(n)$
21: **return** ω[rootnode(ψ)]

INC starts with $\emptyset$ at the *true* node since there is no associated variable.

At each conjunction node, INC takes the union of the child nodes in line 9. Using $n2$ in Figure 1 as an example, if sample drawn at $n4$ is $\omega[n4] = \{\neg z\}$ and at $n5$ is $\omega[n5] = \{y\}$, then unionChild$(\mathsf{Child}(n2), \omega) = \{y, \neg z\}$. At each decision node n, a decision on $\mathsf{Var}(n)$ is sampled from lines 16 to 20. We first calculate the joint probabilities, p_{lo} and p_{hi} of choosing $\neg\mathsf{Var}(n)$ and choosing $\mathsf{Var}(n)$. Subsequently, we sample decision on $\mathsf{Var}(n)$ using a binomial distribution in line 16 with the probability of success being the joint probability of choosing $\mathsf{Var}(n)$. After processing all nodes, the sampled assignment is the output at root node of ψ.

Extending INC ***to*** k ***samples:*** It is straightforward to extend the single sample INC shown in Algorithm 2 to draw k samples in a single pass, where k is a user-specified number. At each node, we have to store a list of k independent copies of partial assignments drawn in ω. At each conjunction node n_c, we perform the same union process in line 9 of Algorithm 2 for child outputs in the same indices of the respective lists in ω. More specifically, if n_c has child nodes c_x and c_y, the outputs of index i are combined to get the output of n_c at index i. This process is performed for all indices from 1 to k. At each decision node n_d, we now draw k independent samples instead of a single sample from the binomial distribution as shown in line 16. The sampling step in lines 16 to 20 are performed independently for the k random numbers. There is no change necessary for the calculation of joint probabilities

in Algorithm 2 as there is no change in literal weights.

Incremental sampling: Given a Boolean formula F and weight function W, INC performs incremental sampling with the sampling process shown in Figure 3. In the initial round, INC compiles F and W into a PROB ψ and performs sampling. Subsequent rounds involve applying a new set of weights W to ψ, typically generated based on existing samples by the controller [10], and performing weighted sampling according to the updated weights. The number of sampling rounds is determined by the controller component, whose logic varies according to application.

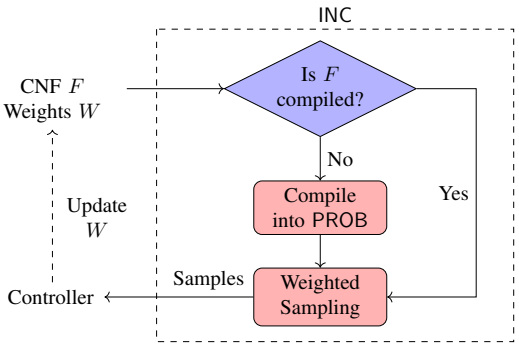


Fig. 3: INC's incremental sampling flow

C. Implementation Decisions

Log-Space Calculations: INC performs *annotation* process - computation of joint probabilities in log space. This design choice is made to avoid the usage of arbitrary precision math libraries, which WAPS utilized to prevent numerical underflow after many successive multiplications of probability values. Using the LogSumExp trick below, it is possible to avoid numerical underflow.

$$\begin{aligned}\log(a+b) &= \log(a) + \log(1+\frac{b}{a}) \\ &= \log(a) + \log(1+\exp(\log(b)-\log(a)))\end{aligned}$$

The joint probability at a decision node n_d is given by $\theta_{\mathsf{Lo}(n_d)} \times$ joint probability of $\mathsf{Lo}(n_d) + \theta_{\mathsf{Hi}(n_d)} \times$ joint probability of $\mathsf{Hi}(n_d)$. Notice that if we were to perform the calculation in log space, we would have to add the two weighted log joint probabilities, termed p_{lo} and p_{hi} in Algorithm 2. Using the LogSumExp trick, we do not need to exponentiate p_{lo} and p_{hi} independently which risks running into numerical underflow. Instead, we only need to exponentiate the difference of p_{lo} and p_{hi} which is more numerically stable. Equations EQ1 and EQ2 can be implemented in log space as follows:

$$Q \text{ of } \wedge\text{-node } n_c = \sum_{c \in \mathsf{Child}(n_c)} Q(c)$$

$$\begin{aligned}Q \text{ of decision-node } n_d = \mathsf{LogSumExp}[&\\ &\log(\theta_{\mathsf{Lo}(n_d)}) + Q(\mathsf{Lo}(n_d)),\\ &\log(\theta_{\mathsf{Hi}(n_d)}) + Q(\mathsf{Hi}(n_d))]\end{aligned}$$

In the equations above, Q refers to the corresponding log joint probabilities in EQ1 and EQ2. In the experiments section, we detail the runtime advantages of using log computations compared to arbitrary precision math computations.

Dynamic Annotation: In existing state-of-the-art weighted sampler WAPS, sampling is performed in two passes - the first pass performs *annotation* and the second pass samples assignments according to the joint probabilities. In INC, we combine the two passes into a single bottom-up pass performing *annotation* dynamically while sampling at each node.

D. Theoretical Analysis

Proposition 1. *Branch parameters of any decision node n_d are correct sampling probabilities, i.e. $W(x_i) : W(\neg x_i) = \theta_{\mathsf{Hi}(x_i)} : \theta_{\mathsf{Lo}(x_i)}$ where $\mathsf{Var}(n_d) = x_i$.*

Proof.

$$\frac{W(x_i)}{W(\neg x_i)} = \frac{\frac{W(x_i)}{W(x_i)+W(\neg x_i)}}{\frac{W(\neg x_i)}{W(x_i)+W(\neg x_i)}} = \frac{\theta_{\mathsf{Hi}(x_i)}}{\theta_{\mathsf{Lo}(x_i)}}$$

We start with the ratio of literal weights of x, multiply both numerator and denominator by $W(x_i)+W(\neg x_i)$ and arrive at the ratio of branch parameters of n_d. Notice that only the ratio matters for sampling correctness and not the absolute value of weights. □

Remark 1. *Let n_d be an arbitrary decision node in* PROB *ψ. When performing sampling according to a weight function W, $\theta_{\mathsf{Lo}(n_d)}$ is the probability of picking $\neg\mathsf{Var}(n_d)$ and $\theta_{\mathsf{Hi}(n_d)}$ is that of $\mathsf{Var}(n_d)$. The determinism property states that the choice of either literal is disjoint at each decision node.*

Proposition 2. INC *samples an assignment τ from* PROB *ψ with probability $\frac{1}{N}\prod_{l\in\tau} W(l)$, where N is a normalization factor.*

Proof. The proof consists of two parts, one for $\wedge$-node and another for decision node.

$\wedge$-node: Let n_c be an arbitrary conjunction node in PROB ψ. Recall that by decomposability property, $\forall c_i, c_j \in \mathsf{Child}(n_c)$ and $c_i \neq c_j$, $\mathsf{VarSet}(c_i) \cap \mathsf{VarSet}(c_j) = \emptyset$. As such an arbitrary variable $x_i \in \mathsf{VarSet}(n_c)$ only belongs to the variable set of one child node $c_i \in \mathsf{Child}(n_c)$. Therefore, assignment of x_i can be sampled independent of x_j where $x_j \in \mathsf{VarSet}(c_j), \forall c_j \neq c_i$. Let τ'_{c_i} be partial assignment for child node $c_i \in \mathsf{Child}(n_c)$. Notice that each partial assignment τ'_{c_i} is sampled independently of others as there are no overlapping variables, hence their joint probability is simply the product of their individual probabilities. This agrees with the weight of an assignment being the product of its components, up to a normalization factor.

Decision node: Let n_d be an arbitrary decision node in PROB ψ and x_d be $\mathsf{Var}(n_d)$. At n_d, we sample an assignment of x_d based on the parameters $\theta_{\mathsf{Lo}(x_d)}$ and $\theta_{\mathsf{Hi}(x_d)}$, which are probabilities of literal assignment by Proposition 1. By Proposition 1, one can see that the assignment of x_d is sampled

correctly according to W. As the sampling process at n_d is independent of its child nodes by the determinism property, the joint probability of sampled assignment of x_d and the output partial assignment from the corresponding child node would be the product of their probabilities. Notice that the joint probability aligns with the definition of weight of an assignment being the product of the weight of its literals, up to a normalization factor.

Since we do not consider the *false* node and treat it as having 0 probability, we always sample from satisfying assignments by starting at the *true* node in bottom-up ordering. Reconciling the sampling process at the two types of nodes, it is obvious that any combination of decision and $\wedge$-nodes encountered in the sampling process would agree with a given weight function W up to a normalization factor $1/N$. In fact, $N = \sum_{\tau_i \in S} W(\tau_i)$ where S is the set of satisfying assignments of Boolean formula F that ψ represents. As mentioned in Proposition 1 proof, normalization factors do not affect the correctness of sampling according to W, and we have shown that INC performs weighted sampling correctly under multiplicative weight functions. □

Remark 2. *From the proof of Proposition 2, the determinism and decomposability property is important to ensure the correctness of* INC. *The smoothness property is important to ensure that the sampled assignment by* INC *is complete. For formula* $F = (x \vee y) \wedge (\neg x \vee \neg z)$*, an assignment* τ_1 *sampled from a non-smooth* PROB *could be* $\{x, \neg z\}$*. Notice that* τ_1 *is missing assignment for variable* y*. By performing smoothing, we will be able to sample a complete assignment of all variables in the Boolean formula as both child nodes of each decision node* n *have the same* $\mathsf{VarSet}(\cdot)$.

V. Experiments

We implement INC in Python 3.7.10, using NumPy 1.15 and Toposort package. In our experiments, we make use of an off-the-shelf KC diagram compiler, KCBox [23]. In the later parts of this section, we performed additional comparisons against an implementation of INC using the Gmpy2 arbitrary precision math package ($\mathsf{INC_{AP}}$) to determine the impact of log-space *annotation* computations.

Our benchmark suite consists of instances arising from a wide range of real-world applications such as DQMR networks, bit-blasted versions of SMT-LIB (SMT) benchmarks, ISCAS89 circuits, and configurable systems [6], [10]. For incremental updates, we rely on the weight generation mechanism proposed in the context of prior applications of incremental sampling [10]. In particular, new weights are generated based on the samples from the previous rounds, resulting in the need to recompute joint probabilities in each round. Keeping in line with prior work, we perform 10 rounds (R1-R10) of incremental weighted sampling and 100 samples drawn in each round. The experiments were conducted with a timeout of 3600 seconds on clusters with Intel Xeon Platinum 8272CL processors.

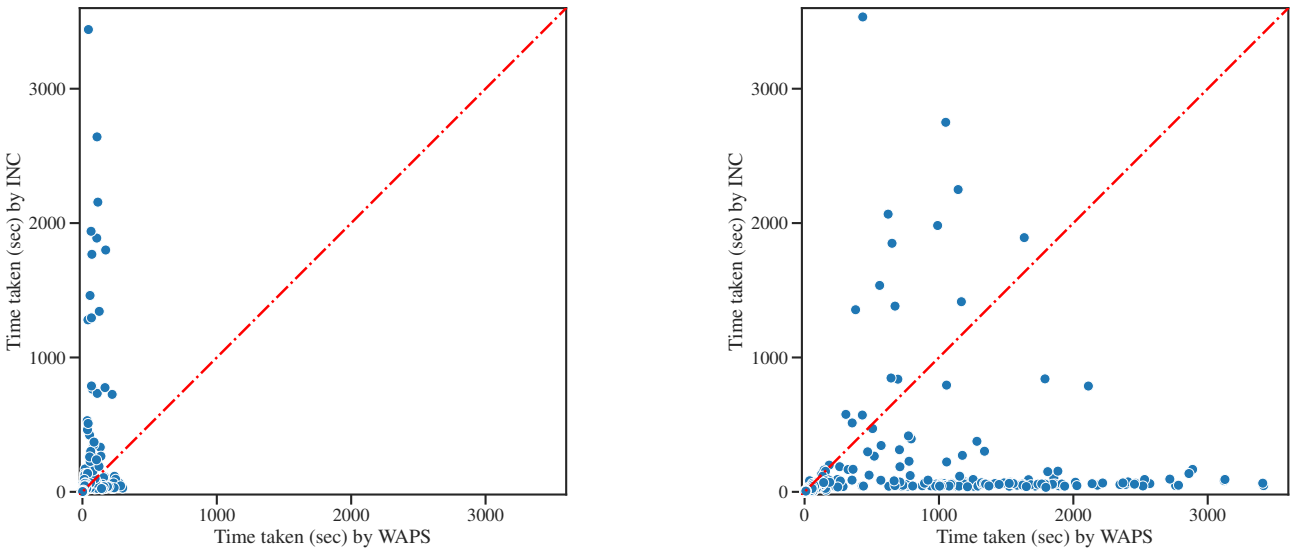
In this section, we detail the extensive experiments conducted to understand INC's runtime behavior and to compare it with the existing state-of-the-art weighted sampler WAPS [6] in incremental weighted sampling tasks. We chose WAPS as it has been shown to achieve significant runtime improvement over other samplers, and accordingly has emerged as a sampler of the choice for practical applications [10]. In particular, our empirical evaluation sought to answer the following questions:

- RQ 1 How does INC's incremental weighted sampling runtime performance compare to current state-of-the-art?
- RQ 2 How does using PROB affect runtime performance?
- RQ 3 How does log-space calculations impact runtime performance?
- RQ 4 Does INC correctly perform weighted sampling?

RQ 1: Incremental Sampling Performance: The scatter plot of incremental sampling runtime comparison is shown in Figure 4, with Figure 4a showing runtime comparison for the first round (R1) and Figure 4b showing runtime comparison over 10 rounds. The vertical axes represent the runtime of INC and the horizontal axes represent that of WAPS. In the experiments, INC completed 650 out of 896 benchmarks whereas WAPS completed 674. INC completed 21 benchmarks that WAPS timed out and similarly, WAPS completed 45 benchmarks that INC timed out. In the experiments, INC achieved a median speedup of $1.69\times$ over WAPS.

Further results are shown in Table I. Observe that for runtime taken for R1 (column 3), WAPS is faster and takes around $0.44\times$ of INC's runtime in the median case. However, INC takes the lead in runtime performance when we examine the total time taken for the incremental rounds R2 to R10 (column 4). For incremental rounds, WAPS always took longer than INC, in the median case WAPS took $4.48\times$ longer than INC. We compare the average incremental round runtime with the first round runtime for both samplers in columns 1 and 2. In the median case, an incremental round for WAPS takes 67% of the time for R1 whereas an incremental round for INC only requires 5.9% of the time R1 takes. We show the per round runtime for 5 benchmarks in Table II to further illustrate INC's runtime advantage over WAPS for incremental sampling rounds, even though both tools reuse the respective KC diagram compiled in R1. This set of results highlights INC's superior performance over WAPS in the handling of incremental sampling settings. INC's advantage in incremental sampling rounds led to better overall runtime performance than WAPS in 75% of evaluations. The runtime advantage of INC would be more obvious in applications requiring more than 10 rounds of samples.

Therefore, we conducted sampling experiments for 20 rounds to substantiate our claims that INC will have a larger runtime lead over WAPS with more rounds. Both samplers are given the same 3600s timeout as before and are to draw 100 samples per round, for 20 rounds. The number of completed benchmarks is shown in Table III In the 20 sampling round setting, INC completed 649 out of 896 benchmarks, timing out on 1 additional benchmark compared to 10 sampling round setting. In comparison, WAPS completed 596 of 896 benchmarks, timing out on 78 additional benchmarks than in



(a) Single Round (R1) Runtime Scatter Plot

(b) Incremental Runtime Scatter Plot

Fig. 4: Runtime comparisons between INC and state-of-the-art weighted sampler WAPS

Statistic	WAPS MEAN(R2 to R10) / WAPS R1	INC MEAN(R2 to R10) / INC R1	WAPS R1 / INC R1	WAPS SUM(R2 to R10) / INC SUM(R2 to R10)	WAPS Total / INC Total
Mean	0.74	0.064	1.03	15.66	6.12
Std	0.24	0.040	1.47	26.42	10.73
Median	0.67	0.059	0.44	4.48	1.69
Max	1.25	0.188	10.65	172.66	73.96

TABLE I: Incremental weighted sampling runtime ratio statistics for WAPS and INC (Numerators and denominators refer to the corresponding runtimes)

the 10 sampling round setting. In addition, WAPS takes on median $2.17\times$ longer than INC under the 20 sampling round setting, an increase over the $1.69\times$ under the 10 sampling round setting.

The runtime results clearly highlight the advantage of INC for incremental weighted sampling applications and that INC is noticeably better at incremental sampling than the current state-of-the-art.

RQ 2: PROB ***Performance Impacts:*** We now focus on the analysis of the impact of using PROB compared to d-DNNF in the design of a weighted sampler. We analyzed the size of both PROB and d-DNNF across the benchmarks that both tools managed to compile and show the results in Table IV. From Table IV, PROB is always smaller than the corresponding d-DNNF. Additionally, PROB is at median $4.64\times$ smaller than the corresponding d-DNNF, and that for PROB is an order of magnitude smaller for at least 25% of the benchmarks. As such, PROB emerges as the clear choice of knowledge compilation diagram used in INC, owing to its succinctness which leads to fast incremental sampling runtimes.

RQ 3: Log-space Computation Performance Impacts: In the design of INC, we utilized log-space computations to perform *annotation* computations as opposed to naively using arbitrary precision math libraries. In order to analyze the impact of this design choice, we implemented a version of INC where the dynamic *annotation* computations are performed using arbitrary precision math in a similar manner as WAPS. We refer to the arbitrary precision math version of INC as $\mathsf{INC_{AP}}$. As an ablation study, we compare the runtime of both implementations across all the benchmarks and show the comparison in Table V. The statistics shown is for the ratio of $\mathsf{INC_{AP}}$ runtime to INC runtime, a value of 1.12 means that $\mathsf{INC_{AP}}$ takes $1.12\times$ that of INC for the corresponding statistics.

The results in Table V highlight the runtime advantages of our decision to use log-space computations over arbitrary precision computations. INC has faster runtime than $\mathsf{INC_{AP}}$ in majority of the benchmarks. INC displayed a minimum of $0.70\times$, a median of $1.12\times$,and a max of $1.89\times$ speedup over $\mathsf{INC_{AP}}$. Furthermore, $\mathsf{INC_{AP}}$ timed out on 2 more benchmarks compared to INC. It is worth emphasizing that log-space

Benchmark	Tool	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total	Speed
or-50-5-5-UC-10	WAPS	**56.6**	56.3	52.5	59.4	52.5	53.6	59.4	53.2	53.4	61.7	**558.6**	1.0×
(100, 253)	INC	1461.3	**7.6**	**8.4**	**8.4**	**8.4**	**8.4**	**8.5**	**8.5**	**8.4**	**8.5**	1536.3	0.4×
or-100-20-9-UC-30	WAPS	**73.0**	69.1	66.7	76.0	66.5	66.9	76.6	66.0	66.9	78.6	706.1	1.0×
(200, 528)	INC	269.5	**4.7**	**4.8**	**4.8**	**4.9**	**5.1**	**4.8**	**4.8**	**4.8**	**5.1**	**313.4**	2.3×
s953a_15_7	WAPS	**1.7**	1.1	1.1	1.2	1.0	1.1	1.2	1.1	1.1	1.3	11.9	1.0×
(602, 1657)	INC	4.9	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**0.7**	**11.5**	1.0×
h8max	WAPS	90.3	104.2	92.4	116.0	94.3	94.1	112.9	92.9	94.4	120.4	1011.9	1.0×
(1202, 3072)	INC	**34.1**	**2.1**	**2.2**	**2.4**	**2.3**	**2.4**	**2.2**	**2.4**	**2.4**	**2.3**	**55.7**	18.2×
innovator	WAPS	195.5	221.9	201.3	244.4	200.1	206.7	247.2	202.0	202.9	257.4	2179.3	1.0×
(1256, 50452)	INC	**32.8**	**1.6**	**1.8**	**1.9**	**1.9**	**1.9**	**1.8**	**1.9**	**1.9**	**1.9**	**49.4**	44.1×

TABLE II: Runtime (seconds) breakdowns for each of ten rounds (R1-R10) between WAPS and INC for benchmarks of different sizes e.g. 'h8max' benchmark consists of 1202 variables and 3072 clauses.

Number of rounds	WAPS	INC
10	674	650
20	596	649

TABLE III: Number of completed benchmarks within 3600s, for 10 and 20 round settings

Statistic	$\frac{\text{WAPS KC size}}{\text{INC KC size}}$
Mean	18.92
Std	81.19
Median	4.64
Max	1734.08

TABLE IV: Statistics for number of nodes in d-DNNF (WAPS KC diagram) over that of smoothed PROB (INC KC diagram).

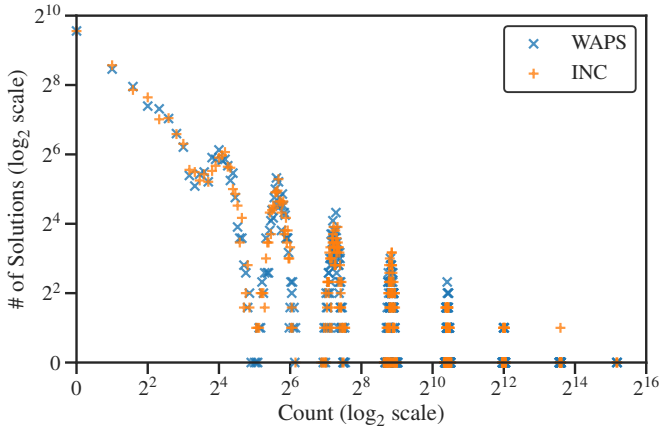


Fig. 5: Distribution comparison for Case110, with log scale for both axes

computations do not introduce any error, and our usage of them sought to improve on the naive usage of arbitrary precision math libraries.

RQ 4: INC ***Sampling Quality****:* We conducted additional evaluation to further substantiate evidence of INC's sampling correctness, apart from theoretical analysis in Section IV-D. Specifically, we compared the samples from INC and WAPS, which has proven theoretical guarantees [6], on the 'case110' benchmark that is extensively used by prior works [4]–[6]. We gave each positive literal weight of 0.75 and each negative literal 0.25, and subsequently drew one million samples using both INC and WAPS and compare them in Figure 5.

Statistic	$\frac{\text{INC}_{\text{AP}}\text{ runtime}}{\text{INC runtime}}$
Mean	1.14
Std	0.16
Median	1.12
Max	1.89

TABLE V: Runtime comparison of INC and INC_{AP}

Figure 5 shows the distributions of samples drawn by INC and WAPS for 'case110' benchmark. A point (x, y) on the plot represents y number of unique solutions that were sampled x times in the sampling process by the respective samplers. The almost perfect match between the weighted samples drawn by INC and WAPS, coupled with our theoretical analysis in Section IV-D, substantiates our claim INC's correctness in performing weighted sampling. Additionally, it also shows that INC can be a functional replacement for existing state-of-the-art sampler WAPS, given that both have theoretical guarantees.

Discussion*:* We demonstrated the runtime performance advantages of INC and the two main contributing factors - a choice of succinct knowledge compilation form and dynamic log-space *annotation*. INC takes longer than WAPS for single-round sampling, mainly because WAPS takes less time for KC diagram compilation than INC, leading to WAPS being faster in single-round sampling. In the incremental sampling setting, the compilation costs of KC diagrams are amortized, and since INC is substantially better at handling incremental updates, it thus took the overall runtime lead from WAPS

in the majority of the benchmarks. Extrapolating the trend, it is most likely that INC would have a larger runtime lead over WAPS for applications requiring more than 10 sampling rounds. The runtime breakdown demonstrates that INC is able to amortize the compilation time over the incremental sampling rounds, with subsequent rounds being much faster than WAPS. In summary, we show that INC is substantially better at incremental sampling than existing state-of-the-art.

VI. Conclusion and Future Work

In conclusion, we introduced a bottom-up weighted sampler, INC, that is optimized for incremental weighted sampling. By exploiting the succinct structure of PROB and log-space computations, INC demonstrated superior runtime performance in a series of extensive benchmarks when compared to the current state-of-the-art weighted sampler WAPS. The improved runtime performance, coupled with correctness guarantees, makes a strong case for the wide adoption of INC in future applications.

For future work, a natural step would be to seek further runtime improvements for PROB compilation since INC takes longer than SOTA for the initial sampling round, due to slower compilation. Another extension would be to investigate the design of a partial *annotation* algorithm to reduce computations when only a small portion of the weights have been updated. It would also be of interest if we could store partial sampled assignments at each node as a succinct sketch to reduce memory footprint, for instance we could store each unique assignment and its count.

Acknowledgement

We sincerely thank Yong Lai for the insightful discussions. Suwei Yang is supported by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd., National University of Singapore, and the Industrial Postgraduate Program (Grant: S18-1198-IPP-II) funded by the Economic Development Board of Singapore. Kuldeep S. Meel is 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), and Ministry of Education Singapore Tier 1 Grant (R-252-000-B59-114).

References

[1] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, pp. 258–265, IEEE, 2007.

[2] M. Jerrum and A. Sinclair, "The markov chain monte carlo method: an approach to approximate counting and integration," 1996.

[3] T. Shi, J. Steinhardt, and P. Liang, "Learning where to sample in structured prediction," in *AISTATS*, 2015.

[4] D. Achlioptas, Z. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *SAT*, 2018.

[5] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge compilation meets uniform sampling," in *LPAR*, 2018.

[6] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "Waps: Weighted and projected sampling," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2019.

[7] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI Mag.*, vol. 28, pp. 13–30, 2007.

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[9] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, p. 307–392, 2019.

[10] E. Baranov, A. Legay, and K. S. Meel, "Baital: An adaptive weighted sampling approach for improved t-wise coverage," in *Proc. 28th European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[11] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. V. den Broeck, K. Kersting, and Z. Ghahramani, "Einsum networks: Fast and scalable learning of tractable probabilistic circuits," in *ICML*, 2020.

[12] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena, "Scalable quantitative verification for deep neural networks," in *Proceedings of International Conference on Software Engineering (ICSE)*, 5 2021.

[13] A. Darwiche and P. Marquis, "A knowledge compilation map," *J. Artif. Intell. Res.*, vol. 17, pp. 229–264, 2002.

[14] Y. Lai, D. Liu, and M. Yin, "New canonical representations by augmenting obdds with conjunctive decomposition," *J. Artif. Intell. Res.*, vol. 58, pp. 453–521, 2017.

[15] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, 1959.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, 1986.

[17] S. ichi Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," *30th ACM/IEEE Design Automation Conference*, pp. 272–277, 1993.

[18] A. Darwiche, "Decomposable negation normal form," *J. ACM*, vol. 48, pp. 608–647, 2001.

[19] A. Darwiche, "A compiler for deterministic, decomposable negation normal form," in *AAAI/IAAI*, 2002.

[20] R. Mateescu, R. Dechter, and R. Marinescu, "And/or multi-valued decision diagrams (aomdds) for graphical models," *J. Artif. Intell. Res.*, vol. 33, pp. 465–519, 2008.

[21] A. Darwiche, "Sdd: A new canonical representation of propositional knowledge bases," in *IJCAI*, 2011.

[22] G. Boole, "An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities," 1854.

[23] Y. Lai, K. S. Meel, and R. Yap, "The power of literal equivalence in model counting," in *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2 2021.